BLEEK · HECHT · LITZKENDORF

Das große Buch zu GFA BASIC Amiga

Bis Version 3.5 und Compiler 3.5
Mit vollständiger Befehlsreferenz zur Systemprogrammierung

DATA BECKER



Wolf-Gideon Bleek
Martin Hecht
Uwe Litzkendorf

Das große GFA-BASIC-Buch zum Amiga

DATA BECKER

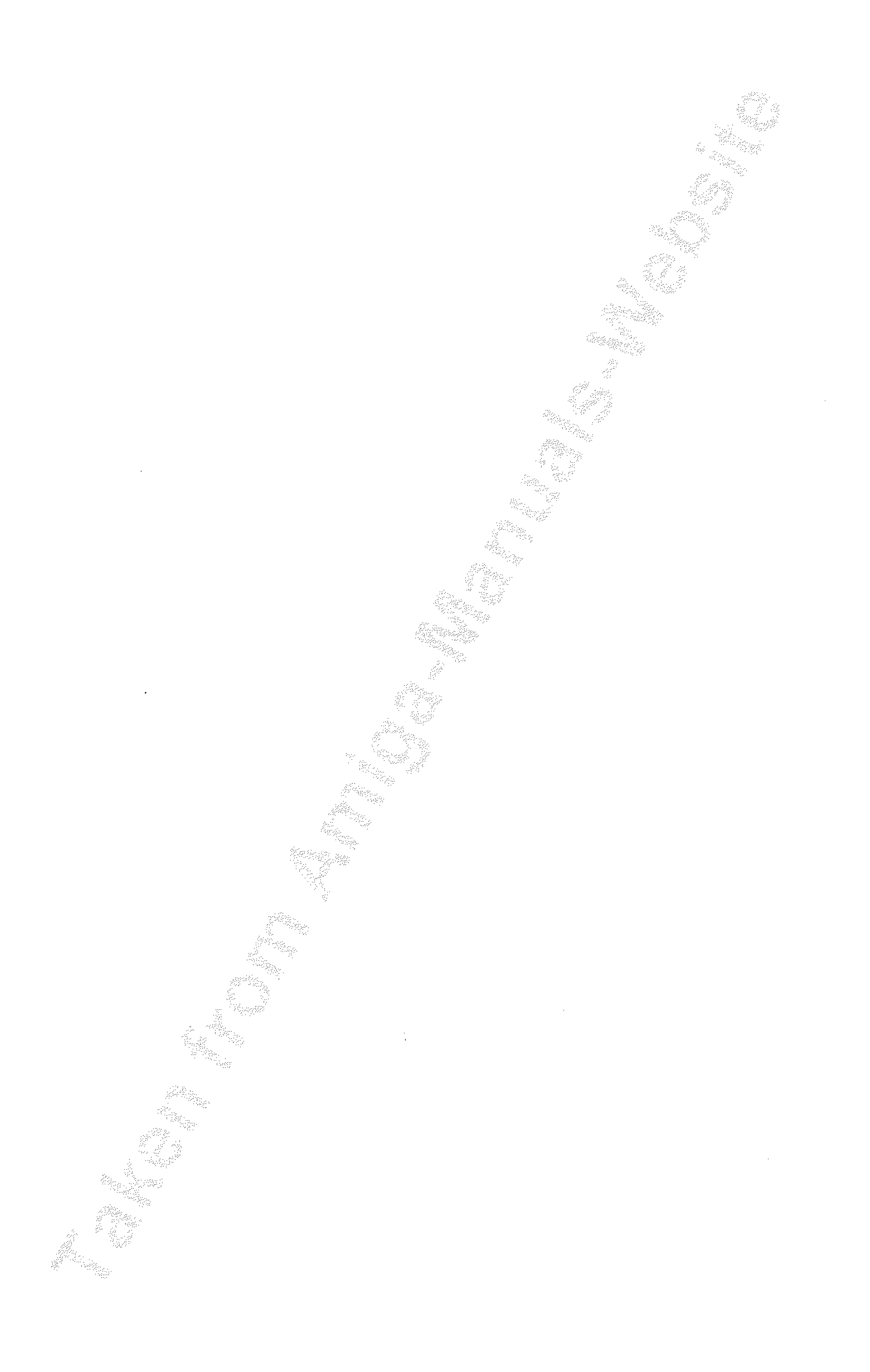
Copyright	© 1990 by DATA BECKER GmbH Merowingerstr. 30 4000 Düsseldorf 1
	1. Auflage 1990
Umschlaggestaltung	Werner Leinhos
Textverarbeitung und Gestaltung	Udo Bretschneider Andreas Quednau
Text verarbeitet mit	Word 5.0, Microsoft
Ausgedruckt mit	Hewlett Packard LaserJet II
Druck und buchbinderische Verarbeitung	Graf & Pflügge, Düsseldorf

ISBN 3-89011-399-0

Wichtiger Hinweis

Die in diesem Buch wiedergegebenen Verfahren und Programme werden ohne Rücksicht auf die Patentlage mitgeteilt. Sie sind ausschließlich für Amateur- und Lehrzwecke bestimmt und dürfen nicht gewerblich genutzt werden.

Alle technischen Angaben und Programme in diesem Buch wurden von den Autoren mit größter Sorgfalt erarbeitet bzw. zusammengestellt und unter Einschaltung wirksamer Kontrollmaßnahmen reproduziert. Trotzdem sind Fehler nicht ganz auszuschließen. DATA BECKER sieht sich deshalb gezwungen, darauf hinzuweisen, daß weder eine Garantie noch die juristische Verantwortung oder irgendeine Haftung für Folgen, die auf fehlerhafte Angaben zurückgehen, übernommen werden kann. Für die Mitteilung eventueller Fehler sind die Autoren jederzeit dankbar.



Vorwort

Endlich! Das GFA-BASIC gibt es jetzt auch für den Amiga.

Um allen Amiga-Usern, die mit dieser äußerst gelungenen Programmiersprache arbeiten möchten, so schnell wie möglich eine kompakte, ausführliche und gut verständliche Dokumentation zukommen zu lassen, haben wir uns überlegt, daß sich hierbei zwei Erfahrungen zusammenbringen lassen. Uwe Litzkendorf, der sicher allen ATARI ST-Besitzern bekannt ist, hatte mit seinem GFA-BASIC-Buch zum ATARI ST einen großen Erfolg. Er ist Kenner dieser Programmiersprache. Martin Hecht, ein Amiga-Programmierer ohne Kompromisse, und Wolf-Gideon Bleek bringt mit vielen Programmierkniffen, großer Erfahrung auf dem Gebiet der strukturierten Programmierung und Betriebssystemkenntnis all sein Wissen ein.

Mit diesem Drei-Autoren-Gespann läßt sich der Inhalt des Buches nur ahnen. Es wurden alle Befehle und Funktionen der Version 3.0 des Amiga-GFA-BASIC dokumentiert, ausprobiert und in Programmbeispielen erklärt. Außerdem bietet der große Grundlagenteil am Anfang des Buches eine gute Voraussetzung für jeden Einsteiger, der neu in der BASIC-Programmierung ist. Ganz besonders freuen wir uns über das letzte Kapitel. Es zeigt den gerade erst erschienenen GFA-BASIC Compiler, der uns freundlicherweise von der Firma GFA-Systemtechnik GmbH, Düsseldorf, schon während der Testphase zur Verfügung gestellt wurde. Ohne die ständigen UpDates hätte dieses Buch nicht den aktuellen Stand haben können, der nun vor Ihnen liegt.

Wir wünschen jedem Leser dieses Buches viel Erfolg beim Lernen dieser neuen Programmiersprache und gutes Gelingen bei eigenen Programmen. Viel Spaß!

Großhansdorf,
im Februar 1990

Wolf-Gideon Bleek
Martin Hecht
Uwe Litzkendorf

Inhaltsverzeichnis

1. Zu diesem Buch

Das Anliegen dieses Buches ist es, die ca. 360 Befehle und Funktionen, die nunmehr vom GFA-V3.03-Interpreter zur Verfügung gestellt werden, nach Schwerpunkten zu ordnen, um so das Auffinden der gesuchten Befehlsbeschreibungen nach problemorientierten Gesichtspunkten zu ermöglichen bzw. zu erleichtern. Um Ihnen eine einheitliche Darstellung zu bieten, haben wir uns an folgende Konventionen gehalten:

Jede Befehls- bzw. Funktionsbeschreibung beginnt mit einer Kopfzeile, die deutlich sichtbar den Befehlsnamen, seine mögliche Abkürzung und eine (sehr knappe) Kurzbeschreibung enthält.

Daran anschließend finden Sie die Syntax, in welcher der Befehl/die Funktion einzusetzen ist. Auf die Beschreibung des Befehls/der Funktion folgt dann gegebenenfalls ein Beispiel oder ein Hinweis auf Beispiele an anderer Stelle.

Innerhalb des Textes wurden für bestimmte Situationen, Vorgaben und Optionen jeweils einheitliche Markierungen benutzt.

< > Wird bei einer Befehlsbeschreibung auf bestimmte Tasten verwiesen, wird ihr Name zur besseren Kenntlichmachung in spitzen Klammern angegeben (z.B. <Shift>, <A>, <Return> oder <Help>).

[ ] Bei Befehlen, deren Syntax variabel ist, wird ein optionaler Befehlsteil in eckigen Klammern angegeben. Dies bedeutet, daß die Angabe (z.B. [;,'] oder [,Länge]) nur dann im Befehl angegeben werden muß, wenn die damit verbundene Option genutzt werden soll.

{ } Viele GFA-BASIC-Befehle können als Abkürzung angegeben werden. Der Interpreter erweitert diese dann selbständig auf die richtige Form. Sollte zu einem Be-

fehl eine Kurzschreibweise existieren, ist diese in der Titelzeile und in der Quick-Referenz innerhalb von geschweiften Klammern angegeben (z.B. { SYS } oder { RET }). Diese geschweiften Klammern werden auch von einigen BASIC-Befehlen (Speicherzugriffe wie CHAR{}, BYTE{} etc.) verwendet. Die Verwechslungsgefahr mit den hier gemeinten Abkürzungsklammern ist jedoch gering.

... Soll eine Folge von Anweisungen innerhalb von Befehlen verdeutlicht werden, geschieht dies anhand einer Punktlinie (z.B. FOR...NEXT).

Grundsätzlich sind in der Syntax-Zeile alle Befehlsnamen in Großbuchstaben, alle Variablen, Parameter und Strings in normaler Schreibweise dargestellt (z.B. OPENW Handle).

Bei den Parameterangaben wurden weitgehend einheitliche Bezeichnungen verwendet:

Adresse	Objektbaumadresse/sonst. Adressen (Adressen werden grundsätzlich als 32-Bit-Integer angegeben).
Anz	Anzahl
Arg	Funktionsargument
Feld	Beliebige Feldbezeichnung
Back=/Var=	Rückgabedaten bei Funktionen
Expr/Expr$	Numerischer/alphanumerischer Ausdruck
Index	Index von Feldelementen
Kanal	Datei-Identifikator
Nummer	GFA-Window-Nummer
Text	Beliebige Zeichenkette
Var/Var$	Beliebiger Variablenname (nicht mit VAR verwechseln!)
Xpos/Ypos	Bildschirmkoordinaten

Bei Dateiname, Programmname und Ordner ist davon auszugehen, daß ein evtl. erforderlicher Suchpfad in den Namen einzubinden ist. Unter dem Begriff Ausdruck (s.o. Expr) wird hier eine beliebige Zusammenstellung von Konstanten, Formeln, Texten, Funktionen und Variablen verstanden, die zusammen ein Ergebnis liefern.

z.B. numerischer Ausdruck:

```
A%=B%+((234^2/4.7)*12.95*C%)^2.1317+@Func(Abc%)
```

z.B. alphanumerischer Ausdruck:

```
A$="Text"+STR$(A%*B%)+SPACE$(10)+@Func$(Abc$)+B$
```

In den Beschreibungen von Funktionen wird nicht explizit angegeben, daß die Ergebnisse aller (auch selbstdefinierter) Funktionen auf verschiedene Weise ausgewertet werden können, z.B. Zuweisung:

```
Var%=@Func    ->  Selbstdefinierte Funktion
Var%=FRE(0)   ->  BASIC-Funktion (z.B. FRE())
```

z.B. Ausgabe:

```
PRINT @Func   ->  Selbstdefinierte Funktion
PRINT FRE(0)  ->  BASIC-Funktion (z.B. FRE())
```

z.B. Abfrage:

```
IF @Func=X    ->  Selbstdefinierte Funktion
IF FRE(0)=X   ->  BASIC-Funktion (z.B. FRE())
```

z.B. Dummy-Aufruf:

```
VOID @Func    ->  Selbstdefinierte Funktion
VOID FRE(0)   ->  BASIC-Funktion (z.B. FRE())
```

In der Syntaxzeile von Funktionen wird in diesem Buch zur Verdeutlichung die Zuweisungsvariante (Var=Funktion()) verwendet. Funktionsaufrufe stehen immer stellvertretend für einen Wert oder String, den diese Funktion liefert. Sie können deshalb

wie jeder beliebige Wert oder String verwendet und eingesetzt werden. Alle Funktionen sind im Anhang unter "Alphabetische Befehlsliste" mit einem vorangestellten (f) gekennzeichnet.

Bei allen Dateizugriffen (außer OUT und INP), die die Angabe einer Kanal-Nummer erwarten, ist die Angabe des Nummernzeichens # optional. In der uns vorliegenden Version V3.0 kann bei allen ON...GOSUB-Name-Befehlen der Teil GOSUB vernachlässigt werden. Er wird vom Interpreter selbständig hinzugefügt (z.B. wird aus ON BREAK Name dann ON BREAK GOSUB Name).

2. Der Amiga

Der MC 68000 von Motorola, der Hauptprozessor der meisten neuen "16-Bitter", wie z.B. Apple Mac, Atari ST, QL, aber eben auch unseres Amiga, bietet im Vergleich zu den früheren 8-Bit-Prozessoren, wie z.B. dem 6502 im C64, eine Vielzahl an Maschinenbefehlen, die erst durch die 16-Bit-Datenbreite verwirklicht werden konnten und gleichzeitig die Fähigkeiten und Geschwindigkeit eines 8-Megahertz-Takters voll zur Geltung brachten.

Was heißt 8 Megahertz? Hertz ist eine aus der Physik bekannte Einheit für Schwingungen pro Sekunde. Fernseher z.B. arbeiten mit einer Bildwiederholungsfrequenz von 50 Hertz. D.h. jede Bildschirmzeile wird innerhalb einer Sekunde 50mal neu aufgebaut. Für unseren Computer bedeutet das, daß ein spezieller Schwing-Quarz mit einer Frequenz von acht Millionen Hertz (achtmillionenmal in der Sekunde!) schwingt und bei jeder Schwingung ein Schalter-Zustand bearbeitet werden kann. In Kombination mit 16 Daten- und 24 Adreßleitungen ergibt sich daraus eine kaum noch zu erfassende Variabilität.

Man stelle sich eine riesige Lagerhalle vor, in der Regale mit insgesamt ca. 16 Millionen (!) Schubladen untergebracht sind. In jeder dieser Schubladen läge eine Information, die bestimmte Auskünfte über die Arbeitssituation im Betrieb gibt. Nun soll jemand innerhalb kürzester Zeit erfassen, welche Information in welcher Schublade liegt und welche Auswirkungen der Inhalt dieser Schublade im Zusammenspiel mit vielen verschiedenen anderen Schubladeninhalten auf die Organisation des Gesamtbetriebes hat.

Das ist, wenn man unser Gehirn als Vergleich nicht in Betracht zieht, unzweifelhaft mit menschlicher Kraft nicht machbar - ein Amiga, ob 2000er, 1000er oder 500er kann das. Da er jedoch auch analysieren, rechnen und einordnen muß, würde dies länger als eine Sekunde dauern. Trotzdem reicht seine Geschwindigkeit aus, um z.B. mit dem GFA-Interpreter in

weniger als einer 20tel Sekunde eine FOR-NEXT-Schleife mit 1000 Schritten zu durchlaufen. Innerhalb eines Schrittes dieser Schleife muß er intern hunderte von Einzelschritten abarbeiten, die Richtigkeit des Programms in seiner Grammatik überprüfen und die verwendeten Befehle analysieren und zuordnen (interpretieren). Wahrhaft eine gewaltige Leistung. War ein 8-Bitter in seinen Kombinationsmöglichkeiten noch einigermaßen überschaubar, so ist ein 16-Bitter real kaum noch zu begreifen.

3. Das GFA-BASIC

Zusammen mit dem Amiga wurde von Anfang an eine Programmiersprache ausgeliefert. Während wir diese bei anderen Home-Computern fest integriert vorfanden, war dies beim Amiga keine Selbstverständlichkeit mehr. Um so erfreuter zeigten sich die Gesichter, als sie vom AmigaBASIC hörten.

Ein wesentlich größerer Befehlsschatz, strukturierte Programmierung und Ausnutzung der Libraries kündeten ein neues, bequemeres Zeitalter der Programmierung an. Jedoch ein Wermutstropfen blieb bei alledem erhalten: die Geschwindigkeit. Das AmigaBASIC vertuschte an keiner Stelle, daß es eine Interpretersprache war, man glaubte teilweise sogar, daß es stolz darauf war.

Sie als Amiga-Besitzer bekommen aber erst jetzt einen Begriff von der Qualität Ihres Computers. Frank Ostrowski stellte vor ca. zwei Jahren sein erstes GFA-BASIC vor. Es gibt wohl kaum andere Programmiersprachen, mit denen auf so atemberaubend einfache Weise selbst schwierige Probleme lösbar sind wie in GFA-BASIC. Überzeugen Sie sich selbst, denn nun gibt es sie auch für den Amiga!

Wir haben nun einen Interpreter, der das Angebot eines MC 68000 in einer für viele nutzbaren Sprache zur Verfügung stellt. Er tritt damit in ernstzunehmende Konkurrenz mit der bis heute favorisierten Compilersprache C. Zugegeben, in manchen Beziehungen werden C- und Assemblerprogrammierung Vorrang behalten. Aber die Gruppe derer, die bereit sind, sich mit der komplizierten Compiler- und Assemblertechnik auseinanderzusetzen, wird sich mehr und mehr in Grenzen halten, da dieses BASIC höchsten Anforderungen mit Sicherheit Genüge tut.

Wie bei jeder Sprache muß man auch hier erst einmal das ABC lernen, um fließend sprechen zu können. D.h., man muß die Grundstrukturen, an denen sich die Sprache orientiert, be-

herrschen, um vom Empfänger (in diesem Fall dem Interpreter) richtig verstanden zu werden. Glücklicherweise haben wir es hier mit BASIC zu tun, dem ja der Ruf anhängt, ein Tausendsassa zu sein, was seine syntaktische Toleranz angeht.

Das GFA-BASIC zwingt - glücklicherweise - zu einer strukturierten Programmierung. Wer sich z.B. mit den Sprachen C, Modula oder Pascal beschäftigt hat, dem werden die Eigenarten der strukturierten Programmierung nichts Neues sein.

In GFA-BASIC wird in jeder Zeile jeweils nur ein Befehl akzeptiert. Außerdem werden die Zeilen vom Interpreter selbsttätig in die entsprechende optische Struktur eingeordnet. Es ist eine wahre Freude zu sehen, wie sauber und ordentlich ein derart durchstrukturiertes Programm hinterher aussieht. Ein weiterer wichtiger Aspekt ist aber, daß dadurch bei der Fehlersuche eine immens große Zeitersparnis eintritt. Befehlszeilen sind ohne Zeichen-Scrolling auf einen Blick erfaßbar. Zusätzlich wird durch das Einrücken der Zeilen sofort erkennbar, z.B. welches ENDIF zu welchem IF oder welches NEXT zu welchem FOR gehört.

Der vielleicht wichtigste Vorteil der Struktur-Programmierung ist aber der, daß beiläufig während der Programmerstellung immer wieder kleine Unterroutinen abfallen, die in sich geschlossen sind und dadurch die Möglichkeit bieten, nach und nach eine umfangreiche Bibliothek an Hilfsprogrammen und allgemein verwendbaren Prozeduren zusammenzustellen. Das wäre prinzipiell in anderen Programmierarten genauso möglich, nur ergibt sich hier die Gelegenheit dazu erheblich seltener. Komfortabel wird es dann noch, wenn man diesen Prozeduren (wie in GFA-BASIC) eine fast beliebig lange Parameterliste übergeben kann. Effektiver geht es fast nicht mehr.

Die V3.0-Version hat gegenüber den früheren Versionen auf anderen Computern erhebliche Veränderungen erfahren. Das beginnt bei der Variablen-Organisation (Einführung von Byte- und Word-Variablen), geht über einen phantastischen Programm-

Editor und endet nach vielen weiteren Änderungen bei wesentlich strafferen Strukturierungsmöglichkeiten (SELECT-CASE, ELSE IF, FUNCTION etc.).

Aus diesen neuen Möglichkeiten ergibt sich eine erhebliche Einsparung an Programmtext und zudem eine ebenso erhebliche Steigerung der Geschwindigkeit im Programmlauf sowie bei der Programm-Entwicklung. Einige neue Befehle ermöglichen eine derart einfache Programmierung auch komplizierter Vorgänge, daß es fast zu einem Kinderspiel wird.

Wahrscheinlich das erste, was Ihnen am neuen GFA-BASIC auffallen wird, ist die rasante Geschwindigkeit des Editors beim Suchen, Ersetzen, Blättern und Scrollen. Diesen Super-Editor kann man schon fast als komplette Textverarbeitung bezeichnen. Ein Suchvorgang durch den gesamten Text dauerte im Durchschnitt nicht länger als ein bis zwei Sekunden(!).

Last but not least sollen hier noch die neuen Editor-Funktionen genannt werden, von denen vor allem <Control><U> (die zuletzt durch <Control><y> gelöschte Zeile restaurieren), die Funktionstasten-Belegung, die interne Zeilennumerierung und der History-Zeilenspeicher im Direktmodus hervorzuheben sind.

Man mag mir vorhalten, daß ich nichts anderes kenne als das GFA-BASIC, was vielleicht zum Teil stimmt, aber nach allem, was ich kenne, ist das V3.0-GFA-BASIC mitsamt seinem Editor die beste Programmiersprache, die es es für den Amiga zu kaufen gibt. Ich bin jedenfalls restlos begeistert und nehme an, daß es den meisten von Ihnen ganz genauso gehen wird.

3.1 Noch einige Anmerkungen zum GFA-BASIC

Zur Erstellung dieses Buches lag uns zuerst eine Testversion vor. Diese besaß schon die Versionsnummer 3.0, hatte aber wesentlich weniger Befehle als die Version 3.0 auf dem Atari ST.

Auch bei weiteren Updates der Testversion wurden es nicht unbedingt so viele Befehle, wie auf dem Atari vorhanden sind. Gleichermaßen traten Probleme bei der Funktion dieser Befehle auf. So waren zwar viele Befehle vom Vorbild übernommen, hatten aber trotz angegebener Dokumentation nicht den Funktionsumfang, wie er beschrieben war.

Auf der anderen Seite gab es auch keine dokumentierten Befehle, die trotzdem implementiert waren, oder vom Interpreter akzeptierte Befehle, die aber nicht bearbeitet wurden.

Sie sehen schon, daß es nicht ganz einfach für uns war, das GFA-BASIC auf dem Amiga zu dokumentieren. Wir haben trotzdem versucht, mit vielen Tricks und Kniffen eine umfangreiche Beschreibung dieser Programmiersprache herauszugeben. Wir hoffen weiterhin, daß bei Ihrer GFA-BASIC-Version die von uns entdeckten Fehler nicht mehr vorhanden sind, so daß Sie ohne solche Sorgen leben können.

Dies wurde bei der Überarbeitung der 1. Auflage bestätigt. Die uns nun vorliegende Version 3.03 hat fast keine Fehler mehr und sogar einige Befehle zusätzlich.

3.2 Zum GFA-Menü

In letzter Minute vor Erscheinen der endgültigen Version wurde das erste Menü ins GFA-BASIC eingefügt. Der Editor kann jetzt also mit einigen Kommandos auch über ein Intuition-Menü und den damit verbundenen Komfort bedient werden. Dieses Menü erreichen Sie, wenn Sie die rechte Maustaste gedrückt halten und mit dem Mauszeiger auf die Titelleiste des Editor-Screens fahren. Es erscheinen 9 Punkte, die hier schnell erklärt werden sollen:

LOAD

Lädt ein GFA-BASIC-Programm. Diese Funktion entspricht dem Punkt LOAD der Leiste in der unteren Zeile und kann auch mit F1 oder <Amiga>-L aufgerufen werden.

SAVE

Speichert ein BASIC-Programm im GFA-Format. Die Funktion gleicht der aus der oberen Zeile der Leiste und kann auch mit <Shift>-F1 oder <Amiga>-S aufgerufen werden.

NEUE NAMEN / New Names

Diese Funktion schaltet die Abfrage nach neuen Variablen-Namen ein oder aus. Sie erhalten sonst bei Verwendung eines neuen Variablen-Namens eine Request-Box, in der nach der Richtigkeit gefragt wird. Dies ist nun unterbunden. Allerdings gilt diese Einstellung nicht für den Direktmodus, da sonst alle alten Variablen-Werte gelöscht würden.

Diese Funktion können Sie über die Tastatur mit <Amiga>-N aufrufen.

RUN

Startet ein Programm. Auch aufzurufen mit <Amiga-R> oder <Shift>-F10.

TASKPRI 0

Setzt die Priorität des GFA-BASIC-Tasks auf 0. Damit wird anderen gleichzeitig laufenden Programmen mehr Zeit des Prozessors zugeteilt. GFA-BASIC wird dadurch langsamer! Diese Funktion kann auch mit <Amiga>-0 aufgerufen werden.

TASKPRI 1

Setzt die Priorität des GFA-BASIC-Tasks auf 1, womit der Interpreter mehr Rechenzeit als andere Programme vom 68000er erhält. So werden die gesamte Programm-Bedienung, der Editor und die Berechnungen beschleunigt. Allerdings werden andere Programme dadurch wesentlich langsamer! Diese Funktion kann auch mit <Amiga>-1 aufgerufen werden.

CLEANUP

Hiermit kann man ohne große Mühe die Sound-Ausgabe, die BOB-Verwaltung und -Bewegung und das Sprite-Handling stop-

pen. Man erspart sich dadurch das Eingeben vieler unterschiedlicher Befehle. Dieser Menüpunkt kann auch mit <Amiga>-C aufgerufen werden.

SAVE ICON

Mit dieser Einstellung können Sie bestimmen, ob das GFA-BASIC zu Ihrem Programm ein Icon auf der Diskette anlegt.

NEWCLI

Startet ein neues CLI auf dem Workbench-Screen. Sie müssen so nicht erst das CLI-Icon auf den Bildschirm holen. Allerdings muß die Shell dann noch von Hand gestartet werden (Version 1.3 der Workbench).

3.3 Der RUN-Only-Interpreter

Zum GFA-BASIC-Paket gibt es einen RUN-Only-Interpreter, der es ermöglicht, Ihr GFA-BASIC-Programm ohne den eigentlichen Interpreter mit Editor laufen zu lassen.

Dies macht es möglich, ein Programm an Freunde oder Bekannte weiterzugeben, die nicht im Besitz des Interpreters sind. Es wird Ihnen von GFA gestattet, diesen RUN-Only-Interpreter beliebig weiterzugeben. So ist es sogar möglich, kommerzielle Programme im GFA-BASIC zu schreiben und diese zu verkaufen, wenn Sie den RUN-Only-Interpreter kostenlos dazu weitergeben.

Nach dem Starten von der Workbench aus erscheint die Auswahl-Box, von der aus Sie den File-Namen des Programms bestimmen können, das gestartet werden soll. Oder aber Sie geben im CLI den File-Namen hinter dem Programm-Aufruf an.

3.4 Der GFA-Editor

Innerhalb des in die Programmiersprache integrierten Editors stehen dem Anwender hilfreiche Funktionen zur Hand, mit

denen die täglichen Korrekturen im Programmtext leichter durchgeführt werden können. So gibt es zum einen das oben schon besprochene Menü, in dem die wichtigsten Befehle leicht erreichbar untergebracht sind. Außerdem kommen noch die Funktionstasten hinzu, die mit den Befehlen belegt sind, die man in der oberen zweizeiligen Leiste wiederfindet.

Wir wollen Ihnen an dieser Stelle einen kurzen Überblick darüber geben, welche Befehle wo zu finden sind.

Die Control-Sequenzen (in alphabetischer Reihenfolge)

ctrl-b	markiert den Blockanfang
ctrl-c	eine Seite weiter
ctrl-cursor-hoch	eine Seite zurück
ctrl-cursor-links	Anfang der Zeile
ctrl-cursor-rechts	Ende der Zeile
ctrl-cursor-runter	eine Seite weiter
ctrl-e	sucht und ersetzt Text
ctrl-shift-e	sucht und ersetzt Text mit vorhergehender Eingabe
ctrl-f	sucht Text
ctrl-shift-f	sucht Text mit vorhergehender Eingabe
ctrl-g	Zeile anspringen
ctrl-home	Programmanfang anspringen
ctrl-k	Blockende markieren
ctrl-n	Leerzeile einfügen
ctrl-q	Blockmenü aufrufen
ctrl-r	eine Seite zurück
ctrl-tab	Tabulator rückwärts
ctrl-u	fügt die gelöschte Zeile (ctrl-y) wieder ein
ctrl-y	löscht eine Zeile
ctrl-z	Programmende anspringen

Funktionstasten

Die beiden Befehlszeilen im oberen Teil des Editor-Screens sind auch über die Funktionstasten zu erreichen. Dabei wird ein Befehl in der unteren Zeile durch einfachen Druck auf eine Funktionstaste und die Befehle in der darüberliegenden Zeile durch die gleichen Tasten in der Kombination mit Shift aufgerufen.

F1	Load (Programm laden)
Shift-F1	Save (Programm speichern)
F2	Merge (ASCII-Programm einbinden)
Shift-F2	Save,A (Programm in ASCII speichern)
F3	Llist (Programm auf dem Drucker ausgeben PTR:)
Shift-F3	Quit (GFA-BASIC beenden)
F4	Block (ruft das Untermenü Block auf)
Shift-F4	New (löscht den Programmtext)
F5	BlkEnd (setzt das Ende der Blockmarkierung)
Shift-F5	BlkSta (setzt den Anfang der Blockmarkierung)
F6	Find (sucht einen Text)
Shift-F6	Replace (sucht und ersetzt einen Text)
F7	Page down (blättert eine Seite tiefer)
Shift-F7	Page up (blättert eine Seite höher)
F8	Insert/Overwr (wechsel zwischen Überschreibe- und Einfügemodus)
Shift-F8	Normal/Interlace (wechsel in der Editor-Screen zwischen 256 und 512 Zeilen)
F9	ClkOn/ClkOff (schaltet die Uhr aus bzw. ein)
Shift-F9	Direct (ruft die direkte Kommandozeile auf)
F10	Test (testet die Programmstruktur)
Shift-F10	Run (startet ein Programm)

Das Blockmenü

Innerhalb des Block-Menüs können alle Befehle auch über ihren Anfangsbuchstaben angesprochen werden.

Copy	Kopiert den Bereich an die Kursor-Position
Move	Verschiebt den Bereich an die Kursor-Position
Write	Schreibt den Bereich im ASCII-Format auf Diskette
Llist	Druck den Bereich (auf PTR:)
Start	Bewegt den Kursor an den Bereichsanfang
End	Bewegt den Kursor an das Bereichsende
^Del	Löscht den Bereich
Hide	Hebt die bestehende Markierung wieder auf

Sondertasten

Taste	Funktion
ESC	ruft die direkte Kommandozeile auf
HELP	versetzt die Editor-Zeile in den alten Zustand Faltet PROCEDUREn/FUNCTIONen auf oder zu
Control-Shift-Alternate	Unterbricht ein laufendes Programm

4. Basis-BASIC

Es ist unmöglich, in einem einzelnen Buch, das die Programmierung in einer bestimmten Programmiersprache erläutern soll, allen Ansprüchen gerecht zu werden. Richtet es sich nach den Interessen der Anfänger, wird es für den Fortgeschrittenen und Profi langweilig. Richtet es sich dagegen nach den Bedürfnissen der Könner, versteht der Anfänger nur noch wenig. Also muß versucht werden, einen Kompromiß zu finden. Dieser besteht darin, dem Anfänger die Grundlagen der Programmierung nahezubringen, ohne in Banalität zu versinken, und komplexe Sachinhalte für Fortgeschrittene darzustellen, ohne in Fachchinesisch abzudriften.

Um nun Anfängern die Möglichkeit zu eröffnen, mit GFA-BASIC den Grundstein zu ihrer Programmierer-Karriere zu legen, will ich hier die wesentlichen Grundlagen dieser Programmiersprache erläutern und zusätzlich eine Einführung in die Computer-Linguistik anbieten. Wer also der Meinung ist, er sei über den Aufbau eines Computers, über Boolesche Logik, Zahlensysteme etc. bereits ausreichend informiert, kann dieses Kapitel vernachlässigen.

Den Einsteigern möchte ich allerdings empfehlen, sich hier mit dem nötigsten Rüstzeug auszustatten, denn ohne gewisse Grundkenntnisse kann man auch den bedienungsfreundlichsten Computer nicht zu sinnvollen Betätigungen bewegen.

Dann wollen wir jetzt einsteigen: Ein Computer ist in erster Linie ein äußerst dummer Zeitgenosse. Ob sich das in Zukunft mit Bio- und Megachips, Transputern, Supraleitern u.ä. wesentlich ändern wird, bleibt abzuwarten. Da Computer der gegenwärtigen Generation nur die beiden Zahlen 0 und 1 unterscheiden können, muß man manchmal gewaltige Anstrengungen unternehmen, um ihre Aufmerksamkeit zu erregen.

Unter normalen Umständen begegnet ein Laie einem Computer mit Skepsis, aber auch mit einer unleugbaren Faszination. Diese Faszination ist der Grund dafür, daß man manchmal vor lauter Ehrfurcht den eigentlich simplen Charakter eines solchen Geräts nicht erkennt. Das einzig Bewundernswerte daran sind die mikroskopische Größe der Schaltungen, die fast unfaßbare Geschwindigkeit, mit der die verschiedenen Operationen durchgeführt werden, und die genial geflochtenen Leiterbahnen auf einem fingernagelgroßen Mikrochip.

In jedem Fall sind es kreative und mit einer äußerst hohen analytischen Intelligenz begabte Menschen, die so ein Ding gebaut haben. Wenn also Ehrfurcht, dann vor den Informatikern, Technikern und Physikern, nicht vor dem Gerät. Wenn Sie nämlich die Stromzufuhr zu den Computer-Prozessoren unterbrechen, ist Ihnen das Gerät hilflos ausgeliefert. Es ist in gewisser Weise sogar sehr wichtig, sich dieses zu vergegenwärtigen, da die Chance, kreativ und produktiv mit einem Computer zu arbeiten, steigt, je mehr man seine Ehrfurcht ihm gegenüber abbaut.

Nach dieser Einleitung nun zur Technik. Es ist hier nicht möglich, in die tieferen Sphären der Computertechnik einzusteigen. Deshalb will ich mich damit begnügen, Ihnen einige Begriffe zu erläutern. Dabei werde ich mich auf solche Begriffe beschränken, die Ihnen in Ihrer Programmierer-Karriere oft begegnen werden und deren Kenntnis zum Verständnis des Computer-Jargons hilfreich ist.

4.1 Computer-ABC

Jeder Computer verfügt über eine zentrale Arbeitseinheit (CPU = Central Processing Unit). Diese ist das eigentliche Herz des Computers. Es handelt sich dabei um einen Prozessor (Arbeits-Chip), der von den o.g. Informatikern so programmiert wurde, daß er selbständig in der Lage ist, eingehende Befehle zu erkennen und auf diese entsprechend zu reagieren. Befehle werden im allgemeinen über die Tastatur eingegeben oder als Programm eingelesen. Im Amiga sind dazu die verschiedenen Schnittstellen

durch ein Bündel von Leitungen mit der CPU verbunden. Dieses Leitungsbündel nennt man Bus. Es gibt Daten- ,Adreß- und Steuerbusse. Während uns der Steuerbus hier nicht näher interessieren soll, sind die Daten- und Adreßbusse doch von erheblicher Bedeutung.

Der Datenbus wird dazu verwendet, Daten (Integer-Binär-Werte) zwischen den Einheiten auszutauschen. So ist es möglich, z.B. einen Wert über die Tastatur an die CPU zu senden, die diesen dann entsprechend der gewünschten Operation verarbeitet und ggf. im Speicher ablegt oder (was eigentlich dasselbe ist) auf dem Monitor ausgibt. Unter dem Speicher versteht man eine Ansammlung von Speicherchips, die ebenfalls durch Busse mit der CPU in Verbindung stehen. Hier kommt der Adreßbus ins Spiel. Um den gesamten Speicher organisieren zu können, wird jedem einzelnen Speicher-Byte (8-Bit-Speicherplatz) eine eigene Adresse zugewiesen. Durch Angabe dieser Adresse ist es also möglich, auf jedes einzelne Byte des Speichers zuzugreifen. Zu den Bits und Bytes kommen wir später. Zunächst sehen wir uns noch einmal die Übertragungsmöglichkeiten per Bus an.

Wie gesagt, der Bus ist ein Bündel von Leitungen. Die CPU verfügt über eine Vielzahl von Pins (Steckfüße des Chips), von denen beim Amiga genau 16 für Daten-Codes und 23 für Adreß-Codes verwendet werden. Diese Pins sind direkt mit den Bussen verbunden. Da in der Digital-Technik eine Stromleitung nur zwei Zustände annehmen kann (an und aus), ist es nicht möglich, über eine einzelne Busleitung andere Werte als 0 (für aus) und 1 (für an) zu senden.

Wenn man sich nun eine Stromleitung vorstellt, in die über einen Schalter Strom eingeleitet wird, und man faßt das freie Ende dieser Leitung an, dann bekommt man einen Schlag. Genauso geht es den Chips, die die jeweilige Information aufzunehmen haben. Aufgrund dieses elektrischen Impulses "weiß" nun der Empfänger, daß ihm etwas Bestimmtes übermittelt werden soll. Er ist darauf programmiert, entsprechend der eintreffenden Informationen einen bestimmten Prozeß auszulösen, auszuführen oder die Information einfach nur zu behalten (speichern). Aber

was kann man schon mit einer einzigen Leitung anfangen, die entweder die Information "Ja" (1=an) oder "Nein" (0=aus) übermitteln kann. Ein Gesprächspartner, der auf die Dauer nur Ja oder Nein sagt, wird schnell langweilig. Man möchte konkretere Auskünfte.

4.2 Bits und Bytes

Dazu benötigen wir mehrere Informationseinheiten (Bit = engl. Abk. für Binary Digit), durch deren Kombination eine Vielfalt an unterschiedlichsten Zuständen ausgedrückt werden kann.

Nimmt man nun zwei Stromleitungen, die unabhängig voneinander an- oder ausgeschaltet werden, sind schon vier verschiedene Kombinationen denkbar. Stellen wir jede stromführende Leitung als 1 und jede "leere" Leitung als 0 dar, dann sieht das so aus:

00	Beide Leitungen führen keinen Strom
10	Leitung 1 = An/Leitung 2 = Aus
01	Leitung 1 = Aus/Leitung 2 = An
11	Beide Leitungen an

Dies ist also schon ein kleiner Schritt mehr in Richtung Kommunikation. Um es noch deutlicher zu machen, wird das Spiel mit vier Stromleitungen wiederholt:

0000	Alle Leitungen aus
1000	Leitung 1 an/2, 3 und 4 aus
0100	Leitung 2 an/1, 3 und 4 aus
0010	Leitung 3 an/1, 2 und 4 aus
0001	Leitung 4 an/1, 2 und 3 aus
1100	Leitung 1 und 2 an /3 und 4 aus
0110	Leitung 2 und 3 an /1 und 4 aus
0011	Leitung 3 und 4 an /1 und 2 aus
1001	Leitung 1 und 4 an /2 und 3 aus
1010	Leitung 1 und 3 an /2 und 4 aus
0101	Leitung 2 und 4 an /1 und 3 aus

1110 Leitung 1, 2 und 3 an/4 aus
0111 Leitung 2, 3 und 4 an/1 aus
1011 Leitung 1, 3 und 4 an/2 aus
1101 Leitung 1, 2 und 4 an/3 aus
1111 Alle Leitungen an

Mit jeder weiteren Leitung verdoppelt sich die Anzahl der Darstellungsmöglichkeiten:

Alle Leitungen aus =	0	
1. Leitung an =	2^0 =	1
2. " an =	2^1 =	2
3. " an =	2^2 =	4
4. " an =	2^3 =	8
5. " an =	2^4 =	16
6. " an =	2^5 =	32
7. " an =	2^6 =	64
8. " an =	2^7 =	128
Summe aller Möglichkeiten, die mit einem BYTE (= 8 BIT) dargestellt werden können =		256 (inkl. Null)

Ich habe diese Liste bewußt mit der Potenz 7 enden lassen, da diese 8 Leitungen mit ihren 256 verschiedenen Aussagemöglichkeiten eine Grundeinheit in der Computerlogik darstellen. Diese Einheit wird Byte genannt. Ein Computer mit einem Arbeitsspeicher von 1 Million Byte kann also einmillionenmal unabhängig voneinander einen solchen Informationsblock (Byte) von je 8 Bit aufnehmen.

Wie Ihnen vielleicht schon bekannt ist, verfügt der Amiga über 256 verschiedene Schriftzeichen. Nach den letzten Ausführungen wissen Sie, daß diese Zahl kein Zufall ist. Warum nun als erste Potenz eine Null gewählt wird, das ist eine mathematische Festlegung. Jeder Wert, der mit dem Wert Null potenziert wird, ergibt den Wert Eins. Diese Eins ist in jedem Zahlensystem die kleinste Einheit.

4.3 Binär-Arithmetik

Der Trick an der Sache ist der, daß jeder Zustand, der sich mit diesen 8 Leitungen darstellen läßt, auch mit einem Wert belegt werden kann. Es wurde nun ein Zahlensystem entwickelt, das ausschließlich die Zustände "An" und "Aus" zur Zahlendarstellung verwendet.

Dabei geht man mathematisch genauso vor, wie wir es von unserem Dezimalsystem her kennen. Der einzige Unterschied ist der, daß als Basis zur Potenz nicht der Wert 10, sondern der Wert 2 genommen wird. Die niedrigste Stelle einer Binärzahl (Bi = griech: zwei) steht ebenso wie in einer Dezimalzahl rechts und die höchste Stelle links.

Als Beispiel nehmen wir ein beliebiges Byte:

```
                          01011101         =  93
Dezimal:       (10 hoch 0)    * 3          =   3
          +    (10 hoch 1)    * 9          =  90
                                      __________
                          Ergebnis         =  93
                                      ==========

Binär   :       (2 hoch 0)    * 1          =   1
          +     (2 hoch 1)    * 0          =   0
          +     (2 hoch 2)    * 1          =   4
          +     (2 hoch 3)    * 1          =   8
          +     (2 hoch 4)    * 1          =  16
          +     (2 hoch 5)    * 0          =   0
          +     (2 hoch 6)    * 1          =  64
          +     (2 hoch 7)    * 0          =   0
                                      __________
                          Ergebnis         =  93
                                      ==========
```

4.4 Das Hexadezimalsystem

Wozu braucht man nun noch das Hexadezimalsystem? Wollte man diese Bytes als Binärzahl darstellen, müßte man dazu je-

desmal eine Zeichenkette von 8 Einsen und Nullen schreiben. Um diese Zahlendarstellung zu vereinfachen, hat man sich die Hexadezimalzahlen ausgedacht.

Dieses Zahlensystem hat alle Eigenschaften der anderen Systeme. Der einzige Unterschied ist, daß als Basis zu den Potenzen weder die 2 noch die 10 genommen wird, sondern der Wert 16. Das hat den Vorteil, daß sich die Hälfte eines Bytes (auch Tetrade oder Nibble genannt), also 4 Bit (maximal darstellbarer Wert = 15), mit einer einzigen Hexadezimalziffer darstellen läßt. Da sich jedoch mit den uns üblicherweise bekannten Zahlen keine größere Zahl als 9 einstellig schreiben läßt, mußten für die Zahlen 10 bis 15 Buchstaben gewählt werden. Der Zahl 10 wird der Buchstabe "A" zugeordnet, der Zahl 11 das "B", 12 = "C", 13 = "D", 14 = "E", und die 15 erhält den Buchstaben "F".

Somit läßt sich also die größte Zahl, die ein Amiga im GFA-BASIC verarbeiten kann, nämlich 2^31-1, mit nur 8 Ziffern darstellen (7FFFFFFF). Im allgemeinen werden Hexadezimalzahlen gekennzeichnet, indem ihnen ein "$" (Dollar, z.B. $1AF7) vorangestellt wird. Das GFA-BASIC handhabt dieses jedoch anders. Hier erhalten Zahlen im Hexa-Format das Kürzel "&H" (z.B. &H1AF7). Hexzahlen haben in erster Linie den Vorteil der Zeit- und Platzersparnis. Weil unser GFA-BASIC aber "international" ist, versteht es auch die "$"-Schreibweise und formt diese in seine richtige Syntax um.

Siehe hierzu auch unter "BIN$, HEX$, OCT$".

4.5 Codes und Opcodes

Ein Wert kann bei entsprechender Vorgabe stellvertretend als numerisches Symbol für ein bestimmtes Wort, Zeichen oder einen Befehl interpretiert werden.

Bei den Schriftzeichen (ASCII-Zeichen = American Standard Code for Information Interchange; amerikanischer Standard-Code für Informationsaustausch) steht z.B. der Wert 65 für den

Buchstaben "A", der Wert 66 für "B", der Wert 67 für "C" usw. (ASCII-Tabelle siehe Anhang). Angenommen, der Wert 192 stünde für "Gebe den im folgenden Byte enthaltenen ASCII-Wert als Schriftzeichen an Cursor-Position auf dem Bildschirm aus". Die Arbeitsweise sieht dann vereinfacht so aus, daß der Amiga weiß, wenn er den Wert 192 empfängt, soll er das ASCII-Zeichen auf dem Bildschirm ausgeben, das dem auf den Befehl folgenden Byte-Wert entspricht.

Der Befehl würde dann so aussehen:

```
Dezimal     :  192      66
Binär       :  11000000 10000010
Hexadezimal:  C0       42
```

Es würde also das Zeichen "B" auf dem Bildschirm ausgegeben werden. Immer dann, wenn wir auf Werte stoßen, die eine Initialisierungsfunktion haben, also als Auslöser für bestimmte Prozesse dienen, haben wir es mit Opcodes (Operation-Codes) zu tun. Solche Opcodes sind nichts anderes als ein Befehl, der symbolisch durch eine Zahl repräsentiert wird. Alles, was den Prozessor interessiert, sind Zahlen, die ihm in verschiedenen Formaten als 'Bit-Häppchen' serviert werden.

Bei der Betriebssystem-Programmierung wurden also verschiedene Funktionsabläufe vordefiniert, die nun z.B. von einem Interpreter anhand solcher Opcodes aufgerufen werden können. Wenn Sie also in BASIC die Zeile

```
PRINT "B"
```

eingeben, wird Ihre Eingabe vom Interpreter auf die oben beschriebene Weise in das Binärformat gewandelt und dem Betriebssystem zur Weiterverarbeitung übergeben. Genausogut könnte man

```
PRINT CHR$(66)        ! (66 = ASCII-Wert für "B")
```

schreiben, nur daß das etwas umständlicher wäre.

4.6 Words und Longwords

Um einen Sprung von den Bytes zu den Words zu machen, braucht man keinen großen Anlauf. Als Word wird eine Informationseinheit bezeichnet, die sich statt aus 8 Bit (1 Byte) aus 16 Bit zusammensetzt. Es können nicht nur 256 verschiedene, sondern 65536 (0+2^0+2^1+2^2+2^3+2^4+2^5+2^6+2^7+2^8+2^9+2^10+2^11+2^12+2^ 13+2^14+2^15) verschiedene Zustände dargestellt werden.

Wie erwähnt, arbeitet der 68000er - die Amiga-CPU - mit einer Datenbreite von 16 Bit. Es können also problemlos Werte im Bereich von 0-65535 (0 bis 2^16-1) von ihm gelesen, verarbeitet und zurückgegeben werden. Doch es gibt noch eine Steigerung. Wenn man nur Werte bis 65535 ausdrücken könnte, könnten manche Leute nicht einmal ihre Steuererklärung damit bearbeiten. Deshalb ist der 68000er so programmiert, daß er (wenn es ihm gesagt wird) zwei 16-Bit-Words direkt nacheinander verarbeitet und dann so vorgeht, als ob es ein 32-Bit-Wert gewesen wäre. Von diesen 32 Bits werden in GFA-BASIC allerdings nur 31 Bits zur Wertedarstellung verwendet. Das letzte (höchste) Bit wird zur Kennzeichnung negativer Zahlen verwendet.

Die Zahlen, die sich damit ausdrücken lassen, dürften für fast jede erdenkliche Steuererklärung ausreichen:

```
2 hoch 31 - 1   =  2147483647
```

Diese 32-Bit-Breite wird als Longword bezeichnet. Die CPU ist so programmiert, daß sie zwischen diesen drei verschiedenen Datenbreiten unterscheiden kann. Ihr ist es also egal, in welchem Format Daten übergeben werden, solange man sagt, welches Format gemeint ist. Bei den BASIC-Befehlen finden Sie deshalb auch POKE/PEEK (Byte-Werte schreiben/lesen), DPOKE/DPEEK (Words schreiben/lesen) und LPOKE/LPEEK (Longwords schreiben/lesen).

4.7 Die Speicherorganisation

RAM

"Random Access Memory" - Freier Zugriffs-Speicher (der Speicherbereich, in den Daten geschrieben und aus dem Daten gelesen werden können).

Das RAM wird noch einmal in Chip-RAM und Fast-RAM unterteilt. Dabei ist das Chip-RAM der Bereich, der von allen Custom-Chips angesprochen werden kann. Diese Chips unterstützen den 68000er bei seiner Arbeit und sind z.B. für Grafik oder Sound-Ausgabe zuständig. Es ist also wichtig, diesen Speicher für solche Daten freizuhalten.

Das Fast-RAM kann nur vom MC 68000 angesprochen werden und ist deshalb schneller, weil keine Taktzyklen für andere Chips vergeben werden müssen. Hier sollten aber nur Programme oder interne Daten stehen, die nicht für Grafik oder I/O benötigt werden.

ROM

"Read Only Memory" - Nur-Lese-Speicher (Speicherchips, in die Daten unveränderbar eingebrannt wurden. z.B.: das Kickstart = im Amiga (500/2000) integriertes Betriebssystem).

Kickstart, das Amiga-Betriebssystem

Der Amiga verwaltet als Multi-Tasking-System seinen Speicher für jedes Programm einzeln. Da dies in zufälliger Reihenfolge geschieht und zusätzlich die Menge des vorhandenen Speichers zwischen 256 und 9728 KByte variieren kann (256 KByte beim Amiga 1000 ohne Speichererweiterung und 9.5 MByte beim Amiga 2000 mit 8-MByte-Karte), läßt sich niemals mit Bestimmtheit sagen, wo welche Speicherbereiche untergebracht sind.

Trotzdem können wir über einige Tatsachen etwas aussagen. So ist der Speicherverbrauch besonders wichtig. Schließlich wollen

Sie ja wissen, wieviel Speicher z.B. übrigbleibt, wenn Sie GFA-BASIC starten. Wir können folgendes festhalten:

Der Workbench-Screen verbraucht mit 2 BitPlanes und einer Auflösung von 640 x 256 Bildpunkten 40 KByte alleine für den Grafik-Speicher. Dazu kommt für jedes geöffnete Window und die darin enthaltenen Icons weitere KBytes, die ausschließlich vom Chip-RAM verbraucht werden, weil nur dort Grafik vom Blitter angesprochen werden kann. Zusätzlich öffnet GFA-BASIC einen Screen, auf dem der Editor sich befindet. Um hier aber nicht noch einmal so viel Speicher zu verbrauchen, wurden nur 2 Farben erlaubt, wodurch sich der Verbrauch auf die Hälfte reduziert.

Und zum Schluß ist GFA-BASIC ein weiterer großer Speicherverbraucher. Allerdings reicht der verbleibende Speicher für alle unsere Demonstrationen aus, denn wir haben darauf geachtet, daß auch 500er-Besitzer in den Genuß unserer Programme kommen.

4.8 Boolesche Logik

Um nun den Computer zu einer Arbeit zu bewegen, die Ergebnisse liefert, mit denen wir etwas anfangen können, mußte ein Verfahren entwickelt werden, das die Arbeitsweise des Computers unserer eigenen angleicht. Es nützt wenig, ihn stur mathematische Aufgaben lösen zu lassen, was ja seine Lieblingsbeschäftigung ist. Man will auch, daß unter bestimmten Bedingungen Entscheidungen von ihm selbständig getroffen werden. Sonst wäre er nichts weiter als ein besserer Taschenrechner.

Ein englischer Mathematiker namens George Boole hat sich dazu eine Form der Arithmetik ausgedacht, die daher auch Boolesche Arithmetik oder Boolesche Algebra genannt wird. Seine Idee war es, für das Grundprinzip menschlicher Entscheidungen allgemeine Regeln zu bestimmen und diese auf den Computer zu übertragen.

Wie entscheidet sich ein Mensch? Entscheidungen sind die Reaktion auf einzelne oder auch auf eine Folge von Bedingungen. Man nimmt eine Situation wahr, ordnet ihre Anforderungen in ein vorhandenes Handlungsschema ein und trifft aufgrund von Übereinstimmungen oder auch Nichtübereinstimmungen mit dem vorhandenen Wertesystem die Entscheidung darüber, was nun zu tun ist. Wir wissen alle, was die Worte "und" und "oder" bedeuten. Beispielsweise könnten Sie folgende Aussagen machen:

1. Wenn es warm ist und ich Zeit habe, werde ich baden gehen.

2. Wenn es kalt ist oder ich keine Zeit habe, werde ich nicht baden gehen.

Wir verknüpfen mehrere Bedingungen miteinander, um danach zu entscheiden, was zu tun oder zu lassen ist. Nichts anderes bewirkt die Boolesche Logik. Dazu hat sich Boole mehrere solcher Verknüpfungsmodi einfallen lassen.

Diese sind:

```
AND, OR, NOT, XOR, IMP, EQV
```

Wobei NOT eine Ausnahme darstellt. Hier handelt es sich nicht um eine Verknüpfung, sondern um die Umkehrung der eingehenden Information. Schauen wir uns an, wie sich diese Verknüpfungen auf die Behandlung der eingehenden Informationen auswirken. Dabei werden nacheinander von rechts ausgehend die Bits des ersten Operanden mit den jeweils gleichrangigen Bits des zweiten Operanden verknüpft.

```
AND:
                11011001          (Byte 1)
          AND   01101101          (Byte 2)
          ---------------
                01001001          Ergebnis

PRINT BIN$(&X11011001 AND &X01101101)
PRINT &X11011001 AND &X01101101
```

Im Ergebnis-Byte wird nur dann ein Bit gesetzt, wenn an der gleichen Stelle im ersten UND im zweiten Ursprungs-Byte ein Bit gesetzt ist.

```
OR:
                 11011001          (Byte 1)
             OR  01101101          (Byte 2)
             ------------
                 11111101          Ergebnis

PRINT BIN$(&X11011001 OR &X01101101)
PRINT &X11011001 OR &X01101101
```

Im Ergebnis-Byte wird immer dann ein Bit gesetzt, wenn entweder im ersten ODER im zweiten ODER in beiden Ursprungs-Bytes ein Bit gesetzt ist.

```
XOR:
                 11011001          (Byte 1)
            XOR  01101101          (Byte 2)
            -------------
                 10110100          Ergebnis

PRINT BIN$(&X11011001 XOR &X01101101)
PRINT &X11011001 XOR &X01101101
```

Im Ergebnis-Byte wird dann ein Bit gesetzt, wenn entweder im ersten ODER im zweiten, ABER NICHT in beiden Ursprungs-Bytes ein Bit gesetzt ist. XOR = exklusiv OR = exklusives (ausschließendes) Oder. Bei den folgenden drei Operationen ist zu beachten, daß Vergleiche jeweils im LONG-Format (32 Bit) durchgeführt werden. Da das gesamte Langwort verknüpft wird, ergeben sich bei diesen Beispielen aufgrund der Zweierkomplementierung Minuswerte (siehe unter Variablen/Boole-Variable).

```
IMP:
                 11011001          (Byte 1)
            IMP  01101101          (Byte 2)
            -------------
11111111111111111111111101101111   Ergebnis

PRINT BIN$(&X11011001 IMP &X01101101)
PRINT &X11011001 IMP &X01101101
```

Im Ergebnis wird immer dann ein Bit gesetzt, wenn in beiden Ursprungs-Bytes ein Bit gesetzt ist oder in beiden Ursprungs-Bytes kein Bit gesetzt ist oder im ersten Byte kein Bit, aber im zweiten Byte ein Bit gesetzt wird. Dies nennt man Implikation. Das Ergebnis ist immer "unwahr" (also 0), wenn im ersten Byte ein Bit gesetzt ist, aber im zweiten keins. In allen anderen Fällen ist das Ergebnis "wahr" (also 1).

```
EQV:
                   11011001          (Byte 1)
               EQV 01101101          (Byte 2)
               ------------
11111111111111111111111101001011    Ergebnis

PRINT BIN$(&X11011001 EQV &X01101101)
PRINT &X11011001 EQV &X01101101
```

Dies ist das Gegenteil zu XOR. Im Ergebnis-Byte wird dann ein Bit gesetzt, wenn in beiden Ursprungs-Bytes entweder ein Bit gesetzt ist oder in beiden Ursprungs-Bytes kein Bit gesetzt ist. Ist in einem der beiden Bytes ein Bit gesetzt, aber im anderen nicht, ist das Ergebnis "unwahr" (also 0).

```
NOT:
               NOT 11011001
               ------------
11111111111111111111111100100110    Ergebnis

PRINT BIN$(NOT &X11011001)
PRINT NOT &X11011001
```

Im Ergebnis-Byte wird dann ein Bit gesetzt, wenn im Ursprungs-Byte kein Bit gesetzt ist. Das Ergebnis ist also immer das Negativ des Ursprungs-Bytes.

Mit Binärmustern lassen sich auch - wie mit Normalzahlen - Additionen, Subtraktionen, Multiplikationen und Divisionen durchführen.

```
Addition:                 11011001       (dez.=217)
                        + 01101101       (dez.=109)
                        ----------
   1.Bit 1+1           =         0       (Übertrag=1)
   2.Bit 0+0+Übertrag=          1        (Übertrag=0)
```

```
  3.Bit 0+1          =          1        (Übertrag=0)
  4.Bit 1+1          =         0         (Übertrag=1)
  5.Bit 1+0+Übertrag=         0          (Übertrag=1)
  6.Bit 0+1+Übertrag=        0           (Übertrag=1)
  7.Bit 1+1+Übertrag=       1            (Übertrag=1)
  8.Bit 1+0+Übertrag=      0             (Übertrag=1)
  9.Bit = Übertrag   =    1
                     ____________
                      101000110          (dez.=326)

  PRINT BIN$(&X11011001+&X01101101)
  PRINT &X11011001+&X01101101

Subtraktion:           11011001          (dez.=217)
                     - 01101101          (dez.=109)
                     ____________
  1.Bit 1-1          =          0        (Übertrag=0)
  2.Bit 0-0          =         0         (Übertrag=0)
  3.Bit 0-1          =        1          (Übertrag=1)
  4.Bit 1-1-Übertrag=        1           (Übertrag=1)
  5.Bit 1-0-Übertrag=       0            (Übertrag=0)
  6.Bit 0-1          =     1             (Übertrag=1)
  7.Bit 1-1-Übertrag=     1              (Übertrag=1)
  8.Bit 1-0-Übertrag=    0               (Übertrag=0)
                     ____________
                        1101100          (dez.=108)

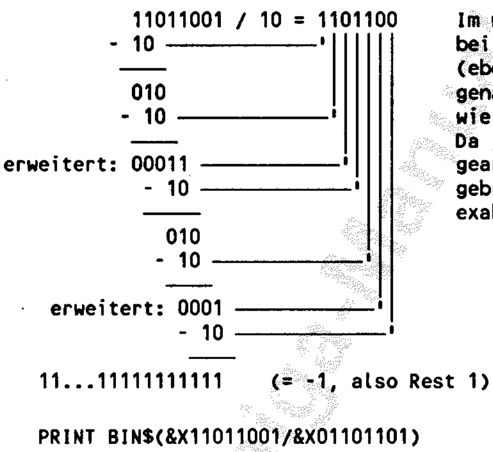
  PRINT BIN$(&X11011001-&X01101101)
  PRINT &X11011001-&X01101101

Multiplikation:    11011001 * 01101101   (dez.: 217*109)
                 _______________________
  OP1/Bit1 * OP2=            01101101    (Rang: 2^0)
+ OP1/Bit2 * OP2            +00000000    (Rang: 2^1)
  Zwischenergebnis =         001101101
+ OP1/Bit3 * OP2           +00000000     (Rang: 2^2)
  Zwischenergebnis =        0001101101
+ OP1/Bit4 * OP2          +01101101      (Rang: 2^3)
  Zwischenergebnis =       01111010101
+ OP1/Bit5 * OP2         +01101101       (Rang: 2^4)
  Zwischenergebnis =      101010100101
+ OP1/Bit6 * OP2        +00000000        (Rang: 2^5)
  Zwischenergebnis =     0101010100101
+ OP1/Bit7 * OP2       +01101101         (Rang: 2^6)
  Zwischenergebnis =    10010111100101
+ OP1/Bit8 * OP2      +01101101          (Rang: 2^7)
                 _______________________
                      101110001100101    Ergebnis

  PRINT BIN$(&X11011001*&X01101101)
  PRINT &X11011001*&X01101101
```

Division (Integer):

Die bitweise Division ist ein relativ kompliziertes Unterfangen. Aus diesem Grund wird hier nur ein einfaches Beispiel behandelt, dessen nähere Erläuterung zu weit führen würde. Es kann hier nur ein ganzzahliges Ergebnis entstehen, da die bitweise Realzahl-Division auf diese Weise nicht machbar bzw. derart aufwendig ist, daß das Thema mehrere Seiten füllen würde.



Im wesentlichen kann bei der Binär-Division (ebenso wie bei +, -, *) genauso verfahren werden wie bei Dezimalrechnungen. Da jedoch in Zweierpotenzen gearbeitet wird, können gebrochene Anteile nicht exakt ermittelt werden.

```
PRINT BIN$(&X11011001/&X01101101)
PRINT &X11011001/&X01101101
```

Bei all diesen Beispielen wurde das Format Byte gewählt, da Ihnen dieses Format am häufigsten begegnen wird. Grundsätzlich sind diese Operationen mit jedem Format durchführbar.

Die eben behandelte Thematik gehört nicht gerade zu den einfachen Dingen in der Computerwelt. Eine Notwendigkeit, mit logischen Operatoren und Bit-Arithmetik umzugehen, besteht allerdings nur für jene, die den Ehrgeiz haben, in die tieferen Ebenen der Programmierung hinabzusteigen. Wenn Sie dazu Neigung verspüren, ist es auf jeden Fall gut, etwas davon gehört bzw. gelesen zu haben.

4.9 Bedingungen und Konsequenzen

In einer anderen Beziehung ist es jedoch ausgesprochen ratsam, sich zumindest mit den beiden Boole-Operatoren AND und OR auseinanderzusetzen. Es gibt nämlich mehrere Befehle im BASIC, die die Angabe einer Bedingung erforderlich machen.

Nehmen wir als Beispiel den Befehl IF...ELSE...ENDIF (siehe dort). Dieses ist wohl der gebräuchlichste Befehl, um den Fortlauf des Programms von einer Bedingung abhängig zu machen. Weiter oben wurden zwei typische Bedingungen und ihre Konsequenzen vorgestellt, wie sie in dieser oder einer ähnlichen Art im täglichen Leben ständig vorkommen.

Wenn es warm ist und ich Zeit habe, werde ich baden gehen. Wenn es kalt ist oder ich keine Zeit habe, werde ich nicht baden gehen. Es werden in beiden Fällen zwei Bedingungen gestellt, deren Erfüllung mit einer Konsequenz verbunden ist.

Um das nun in ein anwendbares Beispiel zur Programm-Entscheidung übertragen zu können, setzen wir für einige Worte in den beiden Sätzen Symbole ein. Für jeden Ausdruck, der etwas bejaht, nehmen wir den Wert 1, und für jeden Ausdruck, der etwas verneint, setzen wir den Wert 0. Also:

```
ist = 1
haben = 1
gehen = 1
nicht haben = 0
nicht gehen = 0
```

In die Struktur einer IF-Bedingung eingefügt, bekommen die beiden Sätze nun folgende Form:

```
If warm=1 And Zeit=1     ! Wenn warm "ist" UND Zeit "haben"
  Baden=1                !  dann baden "gehen"
Endif                    ! Ende der Konsequenz
If Kalt=1 Or Zeit=0      ! Wenn kalt "ist" ODER Zeit "nicht haben"
  Baden=0                !  dann baden "nicht gehen"
Endif                    ! Ende der Konsequenz
```

Sie merken, daß bei Verwendung von Symbolen für "Ja" und "Nein" schon recht komplizierte Entscheidungen möglich sind. Das kann man sogar noch wesentlich weiter führen, wenn man eine weitere Möglichkeit anwendet, die das GFA-BASIC bietet. Man kann mehrere Bedingungen mit einer Klammer zusammenfassen und dazu alternativ weitere Bedingungen stellen. Eine solche Alternative könnte im obigen Beispiel sein:

...oder wenn die Badehalle auf ist,...

Der vollständige Satz wäre dann:

Wenn es warm ist oder wenn die Badehalle auf ist und ich Zeit habe, werde ich baden gehen.

Für die positive Eigenschaft "auf" setzen wir den Wert 1.

```
IF-Struktur:
If (Warm=1 Or Halle=1) And Zeit=1    ! Wenn (warm "ist" ODER
                                     ! Badehalle "auf")
                                     ! UND Zeit "haben"
  Baden=1                            !  dann baden "gehen"
Endif                                ! Ende der Konsequenz
```

Der Faktor "Zeit" bezieht sich hier auf beide vorangestellten Alternativen. Wenn ich z.B. auch in die Badehalle gehen würde, wenn ich keine Zeit hätte, müßte die Klammer anders gesetzt werden, da sich "Zeit" dann nur auf "warm" bezieht:

```
If (warm=1 And Zeit=1) Or Halle=1    ! Wenn (warm "ist" UND
                                     ! Zeit "haben")
                                     ! ODER Badehalle "auf"
  Baden=1                            !  dann baden "gehen"
Endif                                ! Ende der Konsequenz
```

Wenn im vorherigen Beispiel "warm" ODER "Badehalle" die Alternativen waren, so sind es jetzt "(warm UND Zeit)" ODER "Badehalle".

Zur IF-Struktur ist hier zu sagen, daß diese grundsätzlich mit einem ENDIF abgeschlossen werden muß, um dem Interpreter kenntlich zu machen, welche Konsequenzen zu welcher Bedingung gehören.

Es gibt auch die Möglichkeit, eine Alternativ-Konsequenz zu formulieren. Wenn ich sage, daß ich unter bestimmten Bedingungen etwas tun werde, so folgt daraus implizit, daß ich dann, wenn die Bedingungen nicht erfüllt sind, etwas anderes tun werde. Im obigen Beispiel könnte das sein:

Wenn es warm ist oder wenn die Badehalle auf ist und ich Zeit habe, werde ich baden gehen. Andernfalls werde ich nicht baden gehen und ein Buch lesen.

Für "lesen" (positiv) setzen wir hier wieder eine 1 und für "nicht lesen" (negativ) eine 0.

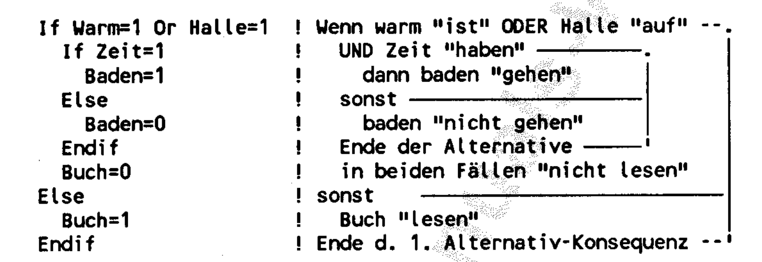
```
IF-Struktur:
If (Warm=1 Or Halle=1) And Zeit=1  ! Wenn (warm "ist" ODER ————.
                                   ! Badehalle "auf")           |
                                   ! UND Zeit "haben"           |
  Baden=1                          !   dann baden "gehen"       |
  Buch=0                           !   dann Buch "nicht lesen"  |
Else                               ! sonst ——————————————————————
  Baden=0                          !   baden "nicht gehen"      |
  Buch=1                           !   Buch "lesen"             |
Endif                              ! Ende d. Alternativ-Konsequenz-'
```

Der Ausdruck ELSE steht hier für "andernfalls" oder auch "sonst". ELSE ist also die konsequente Umkehrung der bei IF gestellten Bedingungen. Die zwischen ELSE und ENDIF eingeschlossenen Konsequenzen bekommen nur dann Gültigkeit, wenn keine der bei IF gestellten Bedingungen zutrifft.

Das heißt wiederum, daß immer dann, wenn das Programm auf eine IF-Abfrage trifft, die mit einer ELSE-Anweisung verbunden ist, entweder die unter IF oder die unter ELSE eingebundenen Konsequenzen Gültigkeit bekommen. Soll das nicht geschehen, wird die ELSE-Anweisung einfach weggelassen. D.h., daß die Nichterfüllung der unter IF gestellten Bedingungen keine weitere Konsequenz hat, als daß das Programm hinter der zugehörigen ENDIF-Anweisung fortgesetzt wird.

Um nun nicht alle Bedingungen, die abgefragt werden sollen, in eine einzige Programmzeile schreiben zu müssen oder um mehrere Folgebedingungen definieren zu können, können solche IF-

Abfragen auch verschachtelt werden. Den Begriff "Ver-schachteln" werden Sie weiter unten auch bei den Schleifen-Strukturen wiederfinden. Damit ist gemeint, daß z.B. in einer IF-Abfrage weitere Abfragen auftreten können, so daß auch Unterverzweigungen möglich werden.



```
If Warm=1 Or Halle=1  ! Wenn warm "ist" ODER Halle "auf" --.
  If Zeit=1           !   UND Zeit "haben" ---------.      |
    Baden=1           !     dann baden "gehen"      |      |
  Else                !   sonst --------------------|      |
    Baden=0           !     baden "nicht gehen"     |      |
  Endif               !   Ende der Alternative -----'      |
  Buch=0              !   in beiden Fällen "nicht lesen"   |
Else                  ! sonst  ----------------------------|
  Buch=1              !   Buch "lesen"                     |
Endif                 ! Ende d. 1. Alternativ-Konsequenz --'
```

Dieses Beispiel entscheidet in zwei Stufen, welche Konsequenzen die Erfüllung zweier unabhängiger Bedingungen haben soll. Erst wenn es warm ist oder die Badehalle auf ist, soll der Zeitfaktor in Betracht gezogen werden. Ist es weder warm noch die Badehalle auf, so wird der Zeitfaktor von vorneherein vernachlässigt und die Entscheidung "Buch lesen" getroffen.

Anhand dieser einfachen Beispiele ist die Übertragbarkeit alltäglicher Entscheidungen in die Logik der Computerwelt hoffentlich etwas deutlicher geworden. Mit Zunahme Ihrer Routine wird auch die Einsicht in die Möglichkeiten dieser Verknüpfungen wachsen. Es ist jedenfalls manchmal recht faszinierend, wie sich durch komplexe Bedingungen unterschiedliche Einflüsse so abfangen und verarbeiten lassen, daß schon der Eindruck eines "intelligenten" Programms entsteht. Die Virtuosität im Umgang mit Bedingungen kennzeichnet den guten Programmierer.

4.10 Flags

Oben wurde noch ein weiteres Prinzip effektiver Programmierung sichtbar. Man nennt es Flags (Flaggen). Diese Flags haben eine sehr wichtige Funktion in jedem Programm, das nicht nur

die Grundfunktionen und -strukturen verwendet, sondern darüber hinaus verschiedene Zustände signalisieren kann, die dann in die Entscheidungsfindung einbezogen werden sollen. Bemühen wir noch einmal unser Beispiel:

```
If Warm=1 Or Badehalle=1
  Flag=1
Endif
.
. ... weiteres Programm ...
.
If Flag=1 And Zeit=1
  Baden=1
Endif
```

Es kann also an irgendeiner Programmstelle ein Zustand ausgewertet werden, dessen Ergebnis erst später zur Wirkung kommen soll. Es ist hier denkbar, daß die in "Flag" gespeicherte Information an mehreren Stellen im Programm ausschlaggebend sein soll.

Um nun nicht an jeder dieser Stellen die Entscheidung treffen (und auch definieren) zu müssen, ob es warm ist oder ob die Badehalle auf ist, kann man diese Entscheidung bei frühestmöglicher Gelegenheit vornehmen und die Information, ob die Entscheidung positiv oder negativ ausgefallen ist, in einer Variablen speichern und diese dann bei weiteren Gelegenheiten abfragen.

Vielleicht weiß ich heute schon, daß es morgen warm sein wird, aber ich weiß heute noch nicht, ob ich morgen Zeit haben werde, baden zu gehen. Also treffe ich die zweite Teilentscheidung erst dann, wenn die dazu erforderlichen Umstände eingetreten sind. Im Beispiel wurde der Variablenname "Flag" willkürlich gewählt. Sie können dafür natürlich jeden beliebigen Variablennamen verwenden.

4.11 Die Variablen

Variablen haben in einem Programm dieselbe Funktion, wie wir sie aus der Mathematik kennen. Sie werden als Platzhalter für Größen oder Ausdrücke (numerische oder alphanumerische) ein-

gesetzt, deren Inhalte erst im Programmverlauf ermittelt und zugewiesen werden und sich im weiteren Programm ständig verändern können. Wir wissen also entweder zum Zeitpunkt der Programmentwicklung nur, "daß" etwas in diesen Variablen abgelegt wird, aber noch nicht "was", oder wir weisen ihnen im Programm-Listing Inhalte zu, die wir an den gegebenen Stellen für notwendig halten.

Um keine Mißverständnisse aufkommen zu lassen: Auf eine Art wissen wir im ersten Fall schon, "was" diese Variablen aufzunehmen haben, wir wissen nur nicht, welcher konkrete Inhalt es sein wird. Denn eine Entscheidung hat man von vorneherein selbst zu treffen und zwar, welcher Variablentyp einzusetzen ist. Es gibt zwei grundlegend verschiedene Typen von Variablen. Das sind die numerischen bzw. Werte-Variablen und die alphanumerischen bzw. Text- oder auch String-Variablen. Numerische Variablen haben die Aufgabe, Werte zu speichern.

```
Hypo=SQR(22^2+13^2)   ! Hypo = Wurzel aus (22 hoch 2 + 13 hoch 2)
```

Ein Beispiel, das wohl alle aus der Schule kennen. Es werden die beiden Kathetenlängen eines rechtwinkligen Dreiecks quadriert und die Quadrate addiert. Anschließend wird nach dem Satz des Pythagoras die Länge der Hypotenuse berechnet. Das Ergebnis dieser Berechnung wird nun der Variablen "Hypo" zugewiesen. Solange keine weiteren Werte an diese Variable übergeben werden, enthält sie die Länge der Hypotenuse des angegebenen Dreiecks. Dieser Wert kann im Laufe des Programms beliebig oft erfragt oder auch durch Neuzuweisungen verändert werden.

Es gibt fünf verschiedene Typen von numerischen Variablen. Wenn wir eine Zahl speichern wollen, die auch Nachkommastellen beinhaltet, also eine sogenannte Realzahl, wird nur der reine Variablenname angegeben. Diesen Typ haben wir im obigen Beispiel kennengelernt. Er benötigt zur Speicherung der ihm übergebenen Werte generell einen Speicherplatz von 8 Byte pro Variable. Dadurch ist eine Genauigkeit von bis zu 13 Stellen möglich. Wertezuweisungen, die über diese 13 Stellen hinausgehen, werden automatisch auf die 13. Stelle gerundet:

```
A=123.125237667231  ergibt  A=123.12523767
```

Bei ganzzahligen Anteilen von mehr als 13 Stellen wird die übergebene Zahl automatisch in das Exponentialformat umgewandelt:

```
A=642653017623.527  ergibt  A=6.4265301762235E+11
```

Andererseits werden Wertzuweisungen, die als "Normal"-Zahl darstellbar sind und im Exponentialformat angegeben wurden, in das Normalformat umgewandelt:

```
A=1284.55E+5        ergibt  A=128455000
```

Im Exponentialformat sind Wertzuweisungen im Bereich von-2.22507385807E-308 bis 3.595386269725E+308 möglich. Eine Exponentialzahl ist folgendermaßen zu lesen:

```
54.6341E+7          entspricht   54.6341 * (10 ^ 7)
```

Ein weiterer Typ ist die Integerzahl. Dies sind Zahlen, die keine Nachkommastellen haben sollen. Es können ihnen also nur Ganzzahlen zugewiesen werden. Um wieder einem Irrtum vorzubeugen: Es können natürlich auch Realzahlen zugewiesen werden. Diese werden in einer Integervariablen nicht als solche gespeichert, sondern die evtl. mit übergebenen Nachkommastellen werden einfach "vergessen":

```
A%=149.523
PRINT A%
==========> Ausgabe:   149
```

Um dem Interpreter klarzumachen, daß er es hier mit einer Variablen des Integertyps zu tun hat, muß dem Variablennamen ein '%' (z.B. Var%) angehängt werden. Pro Variable benötigt dieser Typ einen Speicherplatz von 4 Bytes, woraus sich ein Integer-Bereich von -2147483648 bis 2147483647 ergibt. Ferner gibt es in der Version 3.0 noch die 1- und 2-Byte-Integervariable.

Der dritte numerische Typ ist die Boole-Variable. In ihr können ausschließlich zwei Werte abgelegt werden. Wenn Sie sich später

mit den einzelnen BASIC-Befehlen befassen werden, werden Ihnen mehrere Funktionen begegnen, die als Ergebnis ebenfalls nur zwei verschiedene Werte liefern können. Der eine Wert ist die Null. Dieser Wert gilt im Interpreter als der Wahrheitswert 0. Auch wenn er Wahrheitswert genannt wird, ist dieser Wert stellvertretend für die Feststellung "falsch". Der andere Wahrheitswert ist die Zahl -1. Dieser Wert steht grundsätzlich stellvertretend für die Feststellung "richtig".

Warum für die Feststellung "richtig" eine -1 steht, hat seinen Grund in der sogenannten Zweierkomplement-Darstellung. In einem Longword sind in diesem Fall alle Bits "gesetzt" (angeschaltet), also auch das höchste. Daraus ergibt sich ein Minuswert. Dieses Verfahren hier zu erläutern, würde den gesetzten Rahmen sprengen. Als BASIC-Anfänger muß es Sie im allgemeinen auch nicht näher interessieren. Kluge Köpfe wurden evtl. weiter oben bei Words und Longwords schon stutzig, wo als maximaler Wertebereich 2^31-1 genannt wurde. Dieses liegt an der Zweierkomplementierung, die das oberste Bit als Minus-Identifikator verwendet. Sobald vor einer Zahl ein Minuszeichen steht, wird das oberste Bit eines Longwords gesetzt und die Zahl vorzeichenlos von 2^31 abgezogen. Das daraus entstehende Bit-Muster wird bei der Zweierkomlementdarstellung als Minuswert interpretiert und dementsprechend zurückgegeben.

```
PRINT BIN$(1)
========> Ausgabe: 00000000000000000000000000000001 (1 Bit)

PRINT BIN$(2^31-1)
========> Ausgabe:  1111111111111111111111111111111 (31 Bit)

PRINT BIN$(-2^31)
========> Ausgabe: 10000000000000000000000000000000 (32 Bit)

PRINT BIN$(-2^31+1)
========> Ausgabe: 10000000000000000000000000000001 (32 Bit)

PRINT BIN$(-2^31+2)
========> Ausgabe: 10000000000000000000000000000010 (32 Bit)

PRINT BIN$(-2^31+2^15)
========> Ausgabe: 10000000000000001000000000000000 (32 Bit)
```

```
PRINT BIN$(2^31-2^15)
========> Ausgabe:  1111111111111111000000000000000 (31 Bit)

A%=-1
PRINT LPEEK(VARPTR(A%))
========> Ausgabe: 11111111111111111111111111111111 (32 Bit)
```

Zurück zu den Wahrheitswerten. Nehmen wir dazu als Beispiel den Befehl EXIST. Dieser hat die (oberflächlich gesehen) einfache Aufgabe, festzustellen, ob eine bestimmte Datei auf der Diskette existiert oder nicht. Existiert die Datei, liefert EXIST den Wert -1. Andernfalls wird der Wert 0 zurückgegeben. Andere Werte können von dieser Funktion nicht geliefert werden, weil nur zwei Zustände auftreten können. Entweder die Datei existiert, oder sie existiert nicht. Daraus ist der eigentliche Sinn der Boole-Variablen erkennbar. Überall dort, wo aus einer Entscheidungsfindung nur die Antworten "Ja" oder "Nein" bzw. "richtig" oder "falsch" resultieren können, können Wahrheitswerte in ihnen abgelegt werden.

Die Boole-Variable kann nur einen dieser beiden Werte aufnehmen. Selbst wenn Sie irgendeinen beliebigen Wert zuordnen, wird der Interpreter immer nur zwischen "falsch" (0) und "richtig" (-1) unterscheiden. Alle Werte, die diesem Variablentyp übergeben werden und ungleich 0 sind, werden automatisch als 'wahr', also -1, interpretiert und abgespeichert. Dieser Variablentyp hat den Vorteil, daß er zur Speicherung seiner Inhalte nur 2 Byte pro Variable benötigt. Will man eine Variable als Boole-Variable verstanden wissen, muß man dem Variablennamen das Zeichen '!' (z.B. Var!) anhängen.

In Text- bzw. String-Variablen (String; engl.: Kette/ Reihe/Schnur/ Saite) werden dagegen keine Werte, sondern Textzeichen abgelegt. Genaugenommen sind diese Zeichen ebenfalls Werte, wie wir weiter oben schon kennengelernt haben (ASCII-Zeichen). Nur bei dieser Art der Variablen "weiß" BASIC, daß es die hier abgelegten Werte nicht als Zahlen, sondern als ASCII-Zeichen zu interpretieren hat. Vorausgesetzt, es wurde ihm klargemacht, daß es sich hier um eine String-Variable handelt. Das macht man, indem man dem Variablennamen ein '$' (Dollarzeichen - z.B. Var$) anhängt.

Eine String-Variable kann im GFA-BASIC eine Zeichenkette mit einer Anzahl von 0 bis 32767 einzelner Textzeichen aufnehmen. Das heißt nun nicht, daß jede String-Variable einen Speicherplatz von 32767 Byte (1 ASCII-Zeichen = 1 Byte) reserviert, sondern daß ein String mit maximal 32767 Zeichen übergeben werden kann. Die Länge, die eine solche Variable annimmt, hängt jeweils davon ab, wieviele Zeichen zugeordnet wurden. An Speicherplatz benötigt eine String-Variable soviel Byte, wie Zeichen vorhanden sind. Zusätzlich werden zu jeder String-Variablen noch 6 Byte benötigt.

Zu jeder String-Variablen existiert nämlich ein sogenannter Descriptor (Beschreiber), der sich selbständig die Adresse, also den Standort der Variablen im Speicher, sowie ihre Länge "merkt" (mehr dazu unter ARRPTR und "Variablenorganisation/-typen").

```
A$="BASIC"
PRINT "Der String hat eine Länge von ";LEN(A$);" Zeichen"
PRINT "Die Stringadresse ist ";VARPTR(A$)
PRINT "Der Descriptor für A$ steht bei Adresse ";ARRPTR(A$)
PRINT "Das erste Byte des Strings hat den Wert ";PEEK(Varptr(A$))
PRINT "Der Wert ";PEEK(VARPTR(A$));" repräsentiert das Zeichen ";
PRINT CHR$(PEEK(VARPTR(A$)))
```

Als erstes wurde hier der Variablen "A$" der String "BASIC" übergeben. Anschließend wird mit der BASIC-Funktion LEN die Länge des Strings ermittelt. Um nun in Erfahrung zu bringen, wo das erste Zeichen (Byte) dieses Textausdrucks im Speicher zu finden ist, kann mit der BASIC-Funktion VARPTR (was soviel wie "Variablenzeiger" heißt) die Adresse erfragt werden. Mit der Speicherlese-Funktion PEEK wird nun der Byte-Wert des ersten Zeichens aus der mit VARPTR ermittelten Adresse ausgelesen. Zum Schluß wandelt die Textfunktion CHR$ den so gelesenen Wert wieder zurück in ein Textzeichen, das genau der erste Buchstabe des übergebenen Strings ist.

Zählen Sie zu der mit VARPTR ermittelten Adresse eine 1 hinzu, dann haben Sie die Adresse des zweiten Zeichens. Addieren Sie eine 2, so erhalten Sie die Adresse des dritten Zeichens usw.

```
A$="BASIC"
PRINT "Das zweite Zeichen hat den ASCII-Wert ";PEEK(VARPTR(A$)+1)
```

Wenn Sie den ganzen Variableninhalt auf dem Bildschirm sehen wollen, geben Sie

```
PRINT A$
```

ein, und der String wird auf dem Bildschirm ausgegeben.

4.12 Matrix und Vektor

Das hört sich fast an wie der Titel eines Shakespeare-Dramas. Da es ja nicht langweilig werden soll und man außerdem ohne sie nicht auskommen kann, gibt es noch eine weitere Gattung der Variablentypen. Man nennt sie Felder oder Arrays (array = Aufstellung/Reihe/Ordnung). Wer in der Schule gut aufgepaßt hat, weiß, daß man zur Berechnung einer Funktionskurve mindestens zwei Größen benötigt. In den meisten Fällen werden dies die Größen "X" und "Y" gewesen sein. Der Berechnungsvorgang ist der, daß zu jeder angenommenen Größe "X" anhand einer Funktionsgleichung die Größe "Y" zu ermitteln war. Aus den Schnittpunkten dieser beiden Größen ergaben sich dann die Punkte der Kurve.

Diese beiden Werte stellten auf die jeweilige Funktion bezogen ein Koordinatenpaar dar. Um nun mit den jeweils zusammengehörenden Ordinaten-Werten nicht durcheinanderzukommen, kann man ein Feld einrichten. Dieser Vorgang ist nichts anderes als das, was wohl die meisten unter dem Begriff Wertetabelle kennen. Solch ein zweidimensionales Feld wird auch als Matrix bezeichnet, wovon jede einzelne Dimension einen Vektor darstellt.

Ein Vektor (hier im Sinne von "Einheitsvektor") ist dagegen (im allgemeinen Verständnis) ein Feld, das nur eine Dimension besitzt und meist dazu verwendet wird, mehrere Werte, die zu ei-

ner bestimmten Gruppe gehören, unter einer gemeinsamen "Überschrift" (dem Variablennamen) zusammenfassen und ordnen zu können.

Weitere Informationen zum Umgang mit Feldern finden Sie unter DIM bzw. "Aufbau eines mehrdimensionalen Feldes". Außer den Boole-Variablen benötigen alle anderen Variablentypen in einem Array denselben Speicherplatz, den sie auch als Einzelvariable beanspruchen. Die Boole-Variable benötigt dagegen in einem Array pro Element nur einen Speicherplatz von einem einzigen Bit.

4.13 Erkennungsdienst

Mir ist immer wieder aufgefallen, daß eines der größten Probleme, eine Computersprache zu erlernen, darin besteht, daß man am Anfang in einem Listing nicht unterscheiden kann, was denn nun Befehle (also feststehende Begriffe) und was Namen (also frei bestimmbare Begriffe) sind. Dazu einige Grundregeln.

Die erste: Lernen Sie alle Befehlsnamen so schnell wie möglich auswendig. Alles andere können nur noch freie Begriffe sein! Vorsicht: Ironie, aber auch etwas Wahres ist an diesem banalen Satz dran.

Wenn das so einfach wäre, wie es sich anhört. Gerade bei einer Sprache wie GFA-BASIC, die in der Amiga-Version 3.0 ca. 360 verschiedene Befehle, Funktionen, reservierte Variablen und Felder kennt, ist man schnell überfordert.

In GFA-BASIC kann man in der Amiga-Version problemlos auch Befehlsnamen als Variablennamen verwenden. Es gibt also bis auf reservierte Variablen (TIMER, DATE$ etc.) keine reservierten Begriffe. Bei Prozeduren ist auch schon in anderen Versionen möglich gewesen, zur Namensbildung Befehlsnamen zu verwenden, jedoch lassen sich diese relativ einfach von Befehlen unterscheiden. Eine Prozedur beginnt immer mit der Kennung PROCEDURE, eine Funktion immer mit der Kennung DEFFN,

und ein Label steht immer allein bzw. evtl. mit einem Kommentar (!Kommentar) versehen und endet mit einem Doppelpunkt.

```
Data_label:
```

Oder

```
Xyz_label.1: !Kommentar abc....xyz
```

Prozedur-Aufrufe sind an dem vorangestellten GOSUB oder @ zu erkennen, während Funktionsaufrufe immer mit FN oder ebenfalls mit @ beginnen.

```
Gosub Proc1         oder      @Proc1
Xy%=Fn Funk1        oder      Xy%=@Funk1
```

Bei Labels sind nur solche Bezeichnungen möglich, die vom Interpreter nicht falsch verstanden werden können. Z.B. wird ein Label mit dem Namen Save: in den Befehl SAVE ":" oder der Name Fileselect: in FILES "elect:" umgewandelt.

Der Label-Name Print: ist z.B. also ohne weiteres möglich, sollte jedoch zugunsten der besseren Überschaubarkeit unterbleiben.

Wie vorn schon ausgeführt, haben Variablen (bis auf Real-Variablen ;optional "#") eine Endkennung. Integervariablen erhalten ein '%' (Var%), Byte-Variablen ein "|" (Var|), Word-Variablen ein "&" (Var&), Boole-Variablen ein '!' (Var!) und String-Variablen ein '$' (Var$). Diese sind also ebenfalls an ihren Kennungen leicht auszumachen.

Als Namen können beliebig lange Bezeichnungen eingesetzt werden, die sich aus den normalen Textzeichen (A-Z/a-z/0-9), sowie dem Tiefstrich _ und dem Punkt zusammensetzen können. Bei Namen von Variablen und Funktionen muß das erste Zeichen allerdings ein Buchstabe sein.

```
V.aria_blen_name.1
Feld_titel.xyz%(Dim1,Dim2,...)
1:      <- Soll ein Label sein!
```

```
PROCEDURE 1724_von_a.bis.z
DEFFN hardcopy=X%*Y%
```

Um allen Irritationen aus dem Wege zu gehen, schreibt der BASIC-Editor alle Befehlsnamen grundsätzlich groß. Für den Anfänger ist dies eine gewaltige Hilfe.

Wenn Sie sich einige Zeit mit GFA-BASIC beschäftigt haben, werden Sie diese Hilfe sicher nicht mehr benötigen, da sich allein aus der Logik der Syntax schon eine eindeutige Bestimmung ergibt. Ein Name, hinter dem ein Gleichheitszeichen steht, kann z.B. nur eine Variable sein, und ein Name, dem ein "@" vorangestellt ist und der direkt am Zeilenanfang steht, kann nur ein Prozeduraufruf sein. Befindet sich dagegen vor einem Namen mit vorangestelltem "@" ein Gleichheitszeichen, kann es sich nur um einen Funktionsaufruf handeln usw.

Ich komme wieder auf den obigen "Spruch" zurück. Setzen Sie sich am Anfang zuerst mit den Grundlagen-Befehlen (PRINT, INPUT, READ, DATA, PEEK, POKE, GOSUB etc.) auseinander, und versuchen Sie, die übrigen Komfort-Befehle und -Funktionen erst einmal weitestgehend zu ignorieren. Wenn Sie in den Grund-Befehlen sattelfest sind, erweitern Sie Ihren Sprachschatz nach und nach um die restlichen Befehle.

4.14 Schleifenstrukturen

Schleife nennt man jede Form von Programmstruktur, die bewirkt, daß ein ganz bestimmter Programmblock mehrmals nacheinander durchlaufen wird. In GFA-BASIC sind vier solcher Loops (Loop = Schleife) verwendbar.

1. FOR...NEXT-Schleife
2. DO...LOOP-Schleife
3. REPEAT...UNTIL-Schleife
4. WHILE...WEND-Schleife

Untersuchen wir als erstes zwei Typen, die Bedingungsabfragen implizit verwenden.

Die REPEAT...UNTIL-Schleife

Der Schleifendurchlauf wird durch die Anweisung REPEAT eingeleitet. Im Anschluß an diese Anweisung folgt nun ein beliebig großer Programmblock, der wiederholt ausgeführt werden soll. Die Eigenart dieser Schleife ist eine Bedingungsabfrage am Ende, also am Wendepunkt der Schleife.

Die dort gestellte Bedingung bestimmt, wie oft die Schleife durchlaufen werden soll bzw. unter welchen Bedingungen die Schleife nicht mehr durchlaufen werden soll. Der Schleifenwendepunkt heißt hier UNTIL. Dieser Umkehr-Anweisung wird die genannte Bedingung beigestellt.

```
REPEAT
  INC A
  B=SQR(A)
  PRINT "Wurzel aus ";A;" = ";B
UNTIL B=15 OR A=200
```

Innerhalb der Schleife wird hier ein Zähler (A) durch INC bei jedem Durchlauf um 1 erhöht. Anschließend wird die Wurzel daraus ermittelt, und die beiden Werte werden ausgegeben. Wie Sie sehen, bestimmt die Bedingung B=15, daß die Schleife so oft durchlaufen wird, bis der Wurzelwert mit der Zahl 15 identisch ist.

Wie auch bei IF-Abfragen können hier die Bedingungen mit logischen Operatoren verknüpft werden. So wird hier die Schleife auch (unabhängig von B) verlassen, wenn der Zähler den Wert 200 erreicht.

Die wesentliche Eigenart dieses Schleifentyps ist, daß der Schleifeninhalt auf jeden Fall mindestens einmal durchlaufen wird, da die Bedingungsabfrage erst am Ende der Schleife erfolgt.

Die WHILE...WEND-Schleife

Anders ist es bei der WHILE...WEND-Schleife. Diese wird dagegen gar nicht erst durchlaufen, wenn die Laufbedingung bereits bei Erreichen der Schleife erfüllt ist.

```
A=11
WHILE A<10
  INC A
  PRINT SQR(A)
WEND
```

Der Programmblock innerhalb der Schleife wird nicht ausgeführt. Der Schleifeneinstieg WHILE (während/solange) sagt aus, daß der Block solange durchlaufen werden soll, wie A kleiner als 10 ist. Da A bereits vorher größer ist, wird das Programm sofort hinter dem Wendepunkt WEND fortgesetzt.

Für den Fall, daß die Schleife durchlaufen wird, wird bei jedem Durchlauf geprüft, ob die bei WHILE gestellte Bedingung erfüllt ist. Ist sie das nicht, wird der Block nicht noch einmal ausgeführt und das Programm hinter WEND fortgesetzt. Auch hier können, bei Angabe mehrerer Bedingungen, diese logisch verknüpft werden.

Die FOR...NEXT-Schleife

Eine sehr gebräuchliche Schleifenform, die FOR...NEXT-Schleife, verwendet dagegen keine Bedingung dieser Art, sondern führt die Schleife so oft aus, wie in einer Zählanweisung vorgegeben wird.

```
FOR A%=1 TO 225
  B=SQR(A%)
  PRINT "Wurzel aus ";A%;"  =  ";B
NEXT A%
```

In der FOR-Zeile wird eine beliebige Zählvariable (hier A%) angegeben, die im Verlauf der Schleife solange um den Wert 1 (hier beginnend mit 1) erhöht wird, bis sie den Endwert (hier 225) erreicht hat. Der Schleifenwendepunkt wird durch die Anweisung NEXT gekennzeichnet. Dieser Anweisung ist der Name

der verwendeten Zählvariablen beizustellen. Die FOR...NEXT-Schleife verfügt noch über einige Varianten, die Sie aus der Befehlsbeschreibung zu FOR...NEXT entnehmen können.

Die DO...LOOP-Schleife

Eine Schleife ohne Bedingungsabfrage oder Zähler ist die DO...LOOP-Schleife. Dieser Schleifentyp führt den zwischen DO und LOOP eingeschlossenen Programmblock unendlich lange aus. Es kann keine implizite Abbruchbedingung definiert werden.

```
DO
  INC A%
  B=SQR(A%)
  PRINT "Wurzel aus ";A%;" = ";B
LOOP
```

Soll eine DO...LOOP-Schleife abgebrochen werden, hat man nur die Möglichkeit, entweder die Tasten-Kombination <Control/ Shift/ Alternate> zu drücken oder eine spezielle Abbruchbedingung zu stellen. Diese Abbruchbedingung heißt EXIT IF (siehe dort).

Bei allen Schleifenarten ist es möglich, diese ineinander zu verschachteln. Es kann also in einer Schleife eine weitere, gleich welcher Art, aufgerufen werden.

```
WHILE I%<10
  INC I%
  FOR J=1 TO 10
    REPEAT
      INC K
      PRINT "I% = ";I%;" J = ";J;" K = ";K
    UNTIL K>I%*10
  NEXT J
WEND
```

Bei Verschachtelungen dieser Art ist darauf zu achten, daß die jeweiligen Schleifenwendepunkte (NEXT/UNTIL/LOOP/WEND) in der umgekehrten Reihenfolge ihrer Startanweisungen (FOR/ REPEAT/ DO/WHILE) gesetzt werden.

Falsch:
```
REPEAT
  WHILE A<10
  UNTIL A=10
  WEND
```

Richtig:
```
REPEAT
  WHILE A<10
  WEND
  UNTIL A=10
```

Falsch:
```
FOR I=1 To 10
  FOR J=1 To 10
  NEXT I
  NEXT J
```

Richtig:
```
FOR I=1 To 10
  FOR J=1 To 10
  NEXT J
  NEXT I
```

Sollten Sie diese Reihenfolge nicht einhalten, wird Ihnen der Interpreter beim Programmstart einen "Schwarzen Peter" überreichen.

Diejenigen, die es von anderen BASIC-Interpretern her gewohnt sind, Schleifen anhand von GOTO-Anweisungen zu konstruieren, sollten sich frühzeitig angewöhnen, dafür eine der obigen Schleifen-Konstruktionen einzusetzen. Dieses hat einen wesentlichen Vorteil. GOTO-Anweisungen werden in GFA-BASIC nicht als Struktur-Elemente anerkannt. D.h. also, daß Schleifen nicht auf Anhieb erkennbar wären, während man GFA-Loops sofort an der Zeileneinrückung erkennen kann.

Standard-BASIC :

```
10 A=A+1:Print A;:If A<20 Then Goto 10
```

GFA-BASIC:

```
Label:          ! In diesem Fall würde man
INC A           ! natürlich eine FOR...NEXT-
PRINT A;        ! Schleife verwenden. Hier
IF A<20         ! sollen jedoch nur die
  GOTO Label    ! verschiedenen Strukturen
ENDIF           ! von Schleifen aufgezeigt werden.
```

Besser:

```
WHILE A<20
  INC A
  PRINT A;
WEND
```

oder

```
DO
  INC A
  PRINT A;
  EXIT IF A=>20
LOOP
```

oder

```
REPEAT
  INC A
  PRINT A;
UNTIL A=>20
```

4.15 Vergleichsoperationen

Bei der vorangegangenen Beschreibung der Schleifenstrukturen wurden mehrfach sogenannte Vergleichsoperatoren verwendet.

=	Gleich
==	Ungefähr gleich (28-Bit-Vergleich)
<	Kleiner
>	Größer
<> bzw. ><	Ungleich
<= bzw. =<	Kleiner oder gleich
>= bzw. =>	Größer oder gleich

Diese Operatoren können eingesetzt werden, um zwei Werte oder Textausdrücke miteinander zu vergleichen.

```
PRINT "ABC">"BCD"
```

oder

```
A$="eins"
B$="zwei"
PRINT A$<>B$
```

oder

```
PRINT 123=234
```

oder

```
A%=17356
B=651423.241
PRINT A%<B
```

Diese Beispiele wirken etwas seltsam, erhalten jedoch dann einen Sinn, wenn man weiß, daß als Ergebnis eines Vergleichs immer ein Wahrheitswert (0 oder -1) geliefert wird. Die Beispiele 1 und 3 würden demnach eine Null (falsch) als Ergebnis ausgegeben, da der String ABC nicht größer ist als der String BCD und auch der Wert 234 nicht gleich dem Wert 123 ist. Die Beispiele 2 und

4 liefern dagegen eine -1 (wahr), da A$ und B$ tatsächlich ungleich (<>) und "A%" kleiner als "B" ist.

Auch hier ist eine sinnvolle Anwendung der Boole-Variablen denkbar, indem man die ermittelten Wahrheitswerte an eine solche Variable übergibt.

Sie werden sich evtl. fragen, wie man denn Textausdrücke auf "größer" oder "kleiner" prüfen kann. Das geht folgendermaßen vor sich: Sollen zwei Textausdrücke verglichen werden, geht das BASIC der Reihe nach alle Zeichen der beiden Ausdrücke durch und ermittelt, welches der beiden verglichenen Zeichen den größeren ASCII-Wert besitzt. Der Ausdruck, der in einer der verglichenen Positionen das Zeichen mit dem größeren ASCII-Wert enthält, ist somit der "größere" String. Beim Vergleich zweier Strings auf "kleiner" wird das gleiche Verfahren angewandt. Nur ist hier eben der "kleinere" String der, der zuerst ein Zeichen enthält, dessen ASCII-Wert kleiner ist als der des verglichenen Zeichens des anderen Strings.

Haben die beiden Strings ungleiche Längen und hat der Vergleich bis zum Ende des kürzeren Strings keine Unterschiede zwischen beiden Strings aufgewiesen, so ist beim Vergleich auf "größer" der längere String auch der größere bzw. beim Vergleich auf "kleiner" der kürzere String der kleinere.

Beim Vergleich von Textausdrücken ist allerdings darauf zu achten, daß alle "normalen" Textzeichen einmal in großgeschriebener Form (A-Z/ASCII 65-90) und einmal kleingeschrieben (a-z/ASCII 97-122) existieren. Um also einen tatsächlich gültigen Vergleich auf alphabetische Reihenfolge durchzuführen, müßten beide Textausdrücke vorher vollständig entweder in Groß- oder Kleinschrift übertragen werden (siehe UPPER$).

```
PRINT "ABC"<"ABCD"="abc">"abcd"
```

Dieses fiktive Beispiel ist sicher nicht auf Anhieb zu verstehen. Zuerst werden die ersten beiden Ausdrücke getestet. Da ABC kleiner ist als ABCD, ergibt sich daraus der Wahrheitswert -1. Anschließend ergibt sich aus dem Vergleich abc > abcd der

Wahrheitswert 0. Diese beiden Wahrheitswerte werden nun auf Gleichheit getestet. Da sie ungleich sind, wird der Wert 0 (= falsch) geliefert.

Enthalten die Strings auch andere Zeichen als die von A-Z (Ziffern, Leerzeichen, Satzzeichen), werden diese nach demselben Schema in den Vergleich einbezogen. Siehe dazu auch unter MAX bzw. MIN.

4.16 Vorfahrtsregeln

Richtig heißt es natürlich Vorrangregeln oder Prioritäten. Gemeint ist damit die Reihenfolge, in der die verschiedenen arithmetischen Operationen ausgeführt werden. Aus der Mathematik werden wohl den meisten Konstrukte wie das folgende oder ähnliche bekannt sein:

```
X=Sqr(12^3+(36-22^1.2)-(-4/3))*Sin(13)
```

Formeln dieser Art werden nach folgender Rangfolge aufgelöst:

1. Funktionen: Sqr, Tan, Atn etc. sowie DEFFN-Funktionen.
2. Klammern: ( ) Erst innere, dann äußere Klammern.
3. Potenzierung: ^
4. Negation: -
5. Multiplikation/Division: *, /
6. Modulo/Ganzahldivision: MOD, DIV od. \
7. Addition/Subtraktion: +, -

Da auch Vergleichsausdrücke und logische Verknüpfungen in dieser Form angegeben werden können, werden diese in die Rangfolge mit eingeordnet:

8. Vergleichsoperatoren: =, ==, <>, >, <, =>, <=
9. Logische Operatoren: AND,OR,NOT,XOR,IMP,EQV

```
PRINT (12^3+(36-22^1.2)-(-4/3))*3>14^3
```

oder

```
A=(12^3+(36-22^1.2)-(-4/3))*3 AND 14^3
```

Werden innerhalb eines arithmetischen Ausdrucks nur gleichwertige Operatoren verwendet, werden diese der Reihe nach von links nach rechts ausgeführt. Wollen Sie diese Reihenfolge durchbrechen, können Sie die Berechnungen, die zuerst behandelt werden sollen, in Klammern einfassen. In einer Klammer werden die Operationen wieder in der üblichen Rangfolge bearbeitet.

4.17 Fingerübungen

Im bisherigen Verlauf dieses Kapitels wurden schon einige BASIC-Befehle vorgestellt. Da ihre Verwendung sich aus dem jeweiligen Zusammenhang ergab, gehe ich davon aus, daß hier eine Vorstellung der wichtigsten Grundlagenbefehle und ihrer Verwendungsmöglichkeiten angebracht ist.

Wie in Programmier-Lehrbüchern üblich, besteht das erste Programm eines Computerneulings darin, den Satz "Hello World" auf dem Bildschirm auszugeben. Ich möchte hier nicht nachstehen, wobei ich der Meinung bin, daß in BASIC die Bildschirmausgabe eines beliebigen Textes derart leicht ist, daß ein erheblicher Lernerfolg daraus nicht herzuleiten ist.

Die Initialisierung und Bedienung des Computers wird Ihnen wohl dank der Amiga-Bedienungsanleitung mittlerweile geläufig sein. Wie der GFA-Interpreter gestartet wird bzw. welche Bedienungsfunktionen zu beachten sind, wurde Ihnen ja bereits weiter oben erläutert. Sie haben nun - bis auf die zwei Menüzeilen am oberen Bildrand - einen leeren Editor-Bildschirm vor sich. Der Cursor - das ist das kleine weiße Rechteck oben links

in der Ecke - zeigt Ihnen an, wo bei der nächsten Tastenbedienung ein Schriftzeichen erscheinen wird. Geben Sie nun bitte über die Tastatur ein

```
p "Hello World
```

und drücken Sie anschließend die <Return>-Taste. Hier begegnet Ihnen gleich ein wesentlicher Vorteil des GFA-BASICs. Die meisten Befehle können anhand von Kürzeln eingegeben werden. Sie werden bemerkt haben, daß der Interpreter selbständig die Eingabe auf

```
PRINT "Hello World"
```

erweitert hat. Dieser Komfort hat bei einigen Befehlen erhebliche Wirkung, da es mit Sicherheit wesentlich schneller geht, z.B. statt GRAPHMODE einfach nur "gr" zu schreiben.

Probieren Sie es aus: Schreiben Sie einfach in die nächste Zeile "gr 1", drücken Sie die <Return>-Taste, und schon steht da GRAPHMODE 1. Fahren Sie nun mit der Cursor-Taste Pfeil aufwärts den Cursor eine Zeile aufwärts auf die Graphmode-Zeile, drücken Sie <Control>+<y> gleichzeitig, und schon ist die zweite Zeile auf Nimmerwiedersehen verschwunden.

Außerdem müssen bei Texteingaben nur die Einführungszeichen gesetzt werden, BASIC erkennt das Textende selbständig.

```
PRINT "Text
```

oder

```
A$="Text
```

Wir haben es bei dem GFA-BASIC-Interpreter mit einem äußerst komplexen Programm zu tun, dessen Leistungen erst bei längerer Arbeit deutlich werden.

Ich vergleiche es immer mit einem Qualitätsauto. Der Unterschied zwischen drei bestimmten deutschen Autofirmen und ih-

ren Konkurrenten besteht meistens in den fast unmerklichen Kleinigkeiten - und wenn es nur das Geräusch des Türöffnens ist. Bei der einen Marke tut man es, weil man einsteigen will. Bei der anderen macht man die Tür wieder zu, um sie noch einmal öffnen zu dürfen - das Geräusch ist so schön! Dieses Beispiel ist natürlich stark übertrieben, macht jedoch das Wesen der Qualität deutlich: Auf die Kleinigkeiten kommt es an. Sie werden bei der Arbeit mit GFA-BASIC immer wieder auf solche angenehmen Kleinigkeiten stoßen, und es sollte Ihnen bei der Entwicklung Ihrer eigenen Programme naheliegen, auf diese Kleinigkeiten zu achten.

Aber weiter mit unserem ersten Programm. Sie haben nun mehrere Möglichkeiten, Ihr kleines Programm zu starten.

1. Sie können mit dem Mauszeiger nach rechts oben in die Bildschirmecke auf "Run" fahren und die linke Maustaste drücken.

2. Sie können die <Shift>-Taste und <F10> gleichzeitig drücken.

3. Sie schalten in den Direktmodus um und geben dort den Befehl "ru" (Abk. für "Run") ein, gefolgt von der <Return>-Taste.

In jedem dieser Fälle wird der Bildschirm gelöscht, Ihr freundlicher Gruß oben links auf den Bildschirm geschrieben und eine sogenannte Alert-Box mit dem Hinweis "Programmende" ausgegeben. Nachdem Sie nun die <Return>-Taste gedrückt oder mit dem Mauszeiger auf "Return" geklickt haben, kehrt der Interpreter wieder zum Eingabe-Editor zurück. Damit dürfte eigentlich der Grundstein für Ihre Programmiererkarriere gelegt sein. Alles andere ist nur Übung.

Das Spiel, das wir oben veranstaltet haben, ist im eigentlichen Sinne noch nicht als Programm zu bezeichnen. Bei einer einfachen PRINT-Anweisung handelt es sich lediglich um einen

einzelnen Programmschritt. Erst das sinnvolle Aneinanderreihen von mehreren Programmschritten bildet ein Programm.

Für den Anfang kann es jedoch nützlich sein, sich nach und nach mit den Möglichkeiten vertraut zu machen. Um ein wenig zu probieren, klicken Sie mit dem Mauszeiger im Menükopf des Editors das Feld "Direct" an. Sie sehen jetzt am linken unteren Bildrand die Eingabeaufforderung "OK >". Dies ist der sogenannte Direkt- oder Kommandomodus. Auf dieser Ebene haben Sie nicht die Möglichkeit, Programme zu schreiben, sondern Sie können jeweils nur eine Befehlszeile (max. 255 Zeichen) eingeben, die nach Druck auf die <Return>-Taste direkt ausgeführt wird. Geben Sie bitte ein:

```
OK >p "Hello World
```

Im Ausgabefenster erscheint nun:

```
Hello World
```

Sie sehen, daß jeder Befehl sofort nach <Return> ausgeführt wird. Im Direktmodus haben Sie also eine gute Möglichkeit, erst einmal einige Befehle in Ruhe auszuprobieren. Es gibt zwar Befehle, die hier nicht ausführbar sind (Befehle, die mehr als eine Zeile für ihre korrekte Struktur benötigen), aber zur Übung ist der Direktmodus bestens geeignet. Versuchen Sie es noch einmal:

```
OK >a 1,"Meine erste|Alert-Box",1," Na |toll",a
```

und nun <Return> drücken. So einfach ist das! Wenn Sie genug geübt haben, gelangen Sie durch <Esc> oder durch Drücken der Abbruchtasten <Control/Shift/Alternate> wieder zurück in den Editor.

Wieder im Editor? Gut, dann schreiben Sie nun Ihr erstes richtiges Programm. Das wichtigste an einem Programm sind in jedem Fall die Eingabe- und Ausgabe-Anweisungen. Sicher sind andere Befehle auch wichtig, aber ohne die E/A-Befehle (in Engl.: I/O = Input/Output) geht gar nichts. Also werden wir damit beginnen:

```
REPEAT
  CLS
  INPUT "Wie heissen Sie ";Name$
  INPUT "Wie alt sind Sie ";Alter%
  PRINT "Drücken Sie bitte <Return>"
UNTIL INKEY$=CHR$(13)
PRINT "Ihr Name ist also ";Name$
PRINT "Sie sind ";Alter%;" Jahre alt."
```

Innerhalb der REPEAT...UNTIL-Schleife wird Ihr Name und Alter erfragt. Zuvor wird durch CLS der Bildschirm gelöscht und der Cursor in die linke, obere Bildschirmecke gebracht. Im Anschluß an die Eingaben, die jeweils durch <Return> abzuschließen sind, werden die Daten in Variablen gespeichert: der Name als STRING in einer String-Variablen und das Alter als numerischer Wert in einer Integervariablen. Danach werden Sie durch den PRINT-Befehl aufgefordert, nochmals <Return> zu drücken. In die UNTIL-Bedingungsabfrage wurde ein zweiter Eingabe-Befehl eingebunden, der auf eine einzelne Taste wartet. Wird nun <Return> (ASCII-Wert = 13) gedrückt, wird die Schleife verlassen und Ihre Eingabe - mit einem Kommentar versehen - wieder ausgegeben. Drücken Sie eine andere Taste, wird die Schleife wiederholt, indem zuerst der Bildschirm wieder gelöscht wird und das Spiel von vorn beginnt.

An den beiden abschließenden PRINT-Befehlen ist zu erkennen, daß die Ausgabe variiert werden kann. Durch das Format-Symbol ';' können PRINT-Ausgaben verkettet werden. Dabei ist es egal, in welcher Reihenfolge die Ausdrücke stehen. Eine PRINT-Zeile kann z.B. auch so aussehen:

```
A$="Text"
PRINT LEN(A$)''''A$;" in GFA-";CHR$(66);"ASIC", CRSCOL'INKEY$;
```

Eine solche Zeile kann also sehr bunt gemischt sein. Oben wird zuerst die Variable A$ mit dem Text "Text" initialisiert. Mit LEN(A$) wird die Länge des in A$ gespeicherten Textes, der Inhalt von A$ selbst, ein direkter String-Ausdruck (in GFA-), ein einzelnes ASCII-Zeichen (ASCII 66 = B), wieder ein Direkt-String (ASIC), die aktuelle Cursor-Position (CRSCOL) und zu guter Letzt eine Tastaturabfrage (INKEY$) in die Zeile eingebunden. Während(!) der Textausgabe werden die integrierten

Funktionen bearbeitet, und nach Erledigung wird ihr Ergebnis ebenfalls ausgegeben. So wird die Tastaturabfrage INKEY$ erst bearbeitet, nachdem schon der übrige Text angezeigt wurde. Es sind generell alle Funktionen (ergebnisliefernde Operationen) in einer PRINT-Zeile einsetzbar.

Zur Verkettung der Ausdrücke wurden außerdem noch zwei weitere Format-Symbole verwendet. Ein Komma zwischen den Ausdrücken bewirkt, daß die darauffolgende Ausgabe an der nächsten von 5 Tabulatorpositionen erfolgt (siehe PRINT). Ein Apostroph dagegen gibt nur ein einzelnes Leerzeichen an der Position aus, an der es im Ausdruck steht. Das Semikolon haben Sie oben schon kennengelernt. In diesem Fall wurde es ganz am Ende der Zeile eingesetzt, um zu erreichen, daß die nächste PRINT-Ausgabe (bzw. auch OUT und WRITE) oder INPUT-Eingabe (oder auch INP, INPUT$ etc.) direkt hinter dem zuletzt angezeigten Text positioniert wird. Durch

```
PRINT AT(10,10);
INPUT "Eingabe: ",A$
```

können Sie also auch die Position von Eingabe-Anweisungen bestimmen, wobei jedoch nach erledigter Eingabe der Cursor generell an den Anfang der nächsten Zeile zurückspringt. Im letzten Beispiel habe ich Ihnen wieder zwei neue Varianten vorgestellt. Durch den Zusatz AT(X,Y) kann die Cursor-Position beliebig auf dem Bildschirm bestimmt werden (siehe PRINT). Außerdem habe ich in der INPUT-Zeile ein Komma hinter den Eingabekommentar eingesetzt, um damit zu erreichen, daß die Eingabe ohne Frage- und Leerzeichen direkt hinter dem Kommentar beginnt. Vergleichen Sie:

```
INPUT "Eingabe: ";A$
INPUT "Eingabe: ",A$
```

Unter den Befehlsbeschreibungen finden Sie noch weitere E/A-Befehle, die an Möglichkeiten fast keine Wünsche mehr offen lassen. Versuchen wir uns nun ein wenig an der Grafik.

Möchten Sie lieber Kreise oder Rechtecke zeichnen? Oder vielleicht eine Freihandlinie? Alles kein Problem! Fortschrittliche BASIC-Dialekte haben selbstverständlich ein reichhaltiges Angebot an grafischen Befehlen. Das ist ganz und gar nicht so selbstverständlich, wie es uns heutzutage erscheint. Vor nur wenigen Jahren mußten die Algorithmen dazu noch per Hand geschrieben werden - und wer weiß schon aus dem Stegreif, wie ein Kreis oder eine Ellipse konstruiert wird.

Wir haben es da einfach. In der Befehlsliste im Anhang finden Sie eine Fülle von Grafikbefehlen, deren Bedienung denkbar einfach ist. Unser Bildschirm ist in Hires (High Resolution = hohe Auflösung) in 640 Bildpunkte horizontal und 256 Bildpunkte vertikal eingeteilt. Der Bildpunkt mit der Koordinate 0/0 liegt üblicherweise (siehe OPENW) in der äußersten linken, oberen Bildschirmecke. Durch dieses Wissen läßt sich nun jeder einzelne Bildschirmpunkt auch einzeln benennen. Sie müssen sich also nur entscheiden, an welcher Bildschirmposition Sie ein Grafik-Objekt sehen möchten, übergeben dem Befehl die entsprechenden Koordinaten, und schon erscheint das Objekt. Die folgende Zeile zeichnet eine Ellipse, deren Zentrum ziemlich genau in der Mitte des Bildschirms liegt:

```
ELLIPSE 310,120,300,110
```

Die ersten beiden Parameter bestimmen die Position, der dritte den horizontalen Radius und der vierte den vertikalen Radius der Ellipse. Diese Koordinatenangaben beziehen sich - wie gesagt - auf Hires. Beim Amiga gibt es aber zwei Auflösungsstufen. Die zweite ist Lowres (Low Resolution = niedrige Auflösung) mit 320 horizontalen und 256 vertikalen Punkten. Mit diesen Informationen und den Beschreibungen der Befehle werden Sie sicherlich keine größeren Probleme mit der Grafik haben.

Malen Sie doch mal ein kleines Bild mit dem folgenden Programm.

```
DEFFILL ,2,4              ! Füllmuster einstellen
DO                        ! Hauptschleifenstart ———————.
  REPEAT                  ! 1. REPEAT-Schleife ————.   |
  UNTIL MOUSEK            ! wartet auf Mausklick ——'   |
  MOUSE x,y,k             ! Mauskoordinaten in X,Y     |
  REPEAT                  ! 2. REPEAT-Schleife ————.   |
    MOUSE xx,yy,k         ! Neue Koordinaten in Xx,Yy  |
    IF MOUSEK=1           ! Linke Maustaste?       |   |
      LINE x,y,xx,yy      ! Dann Linie zeichnen    |   |
      x=xx                ! Neue Koordinaten werden|   |
      y=yy                ! Alte Koordinaten       |   |
    ENDIF                 !                        |   |
    IF MOUSEK=2           ! Rechte Maustaste?      |   |
      flag!=TRUE          ! merken                 |   |
      GRAPHMODE 3         ! Grafikmodus XOR        |   |
      BOX x,y,xx,yy       ! Box zeichnen           |   |
      BOX x,y,xx,yy       ! Box löschen            |   |
      GRAPHMODE 1         ! Grafikmodus Replace    |   |
    ENDIF                 !                        |   |
    IF MOUSEK=3           ! Beide Maustasten?      |   |
      flag!=FALSE         ! Merker löschen         |   |
      FILL xx,yy          ! Ab Mausposition füllen |   |
    ENDIF                 !                        |   |
  UNTIL MOUSEK=0          ! Maustaste losgelassen? --'  |
  IF flag!=TRUE           ! Merker für Maustaste 2?    |
    flag!=FALSE           ! Merker löschen             |
    BOX x,y,xx,yy         ! Box nochmal zeichnen       |
  ENDIF                   !                            |
LOOP                      ! Zurück zum Anfang ————————'
```

Hier finden Sie gleich ein Beispiel für den Einsatz von Flags. Flag! ist hier eine völlig beliebige Boole-Variable, die die Aufgabe hat, sich zu merken, ob in der 2. REPEAT-Schleife die rechte Maustaste gedrückt wurde. Wurde sie das, wird im XOR-Modus zweimal eine Box auf dieselbe Stelle gezeichnet. Das hat den Effekt, daß die Box bei jedem Durchlauf nur für einen kurzen Moment zu sehen ist. Wird die rechte Maustaste wieder losgelassen, wird anschließend die Box noch einmal im REPLACE-Modus (siehe GRAPHMODE) gezeichnet.

Achten Sie bitte darauf, daß Sie, wenn Sie den Füllprozeß mit beiden Maustasten gleichzeitig auslösen wollen, zuerst die rechte und dann die linke Taste drücken, da sonst im Linienmodus schon ein Punkt gezeichnet wird, ehe der Füllprozeß beginnt. Dazu eine kleine Spielerei:

```
PLOT 320,128                   ! Linienstart
FOR i=0 TO 200 STEP 0.1        ! 10tel-Step-Schleife ——.
  ADD a,0.1                    ! Radius vergrößern      |
  x%=320+COS(i)*a*1.6          ! X-Koordinate           |
  y%=128+SIN(i)*a              ! Y-Koordinate           |
  DRAW  TO x%,y%               ! Linie zeichnen         |
NEXT i                         ! Nächster Schritt ——————'
```

Beachten Sie hier, daß die FOR...NEXT-Schleife mit einer Realvariablen (I) läuft, da Integervarialen nur Ganzzahlen aufnehmen können. Mit einer Integer-Zählvariablen würde das Programm lediglich eine gerade Linie zeichnen und unendlich weiterlaufen, da die Zählvariable immer wieder auf Null gesetzt würde. Die Linie ergibt sich dann ausschließlich aus der Erhöhung des Radius durch den Real-Faktor "A".

Soweit zur Einführung in die Programmiersprache GFA-BASIC und in die Hintergründe und Grundlagen der Computer-Programmierung. Es ist mir nicht immer leicht gefallen, die richtigen Worte zu finden, um ein Thema auch - und gerade - für den Computer-Anfänger verständlich und durchschaubar werden zu lassen. Ich habe es jedenfalls nach besten Kräften versucht und hoffe, daß Ihnen dieses Rüstzeug auf Ihrem weiteren Weg in der "hohen Schule" der Programmierkunst ein Stück helfen wird.

Sie finden zudem im Verlauf des Buches immer wieder kleine Einschübe, die dazu beitragen sollen, Ihre hier erworbenen Kenntnisse zu vertiefen. Noch ein kleiner Tip zum Schluß. Setzen Sie sich in der Anfangszeit so viel wie möglich mit fremden Programmen auseinander. Achten Sie dabei darauf, daß diese in ihrer Komplexität ungefähr Ihrem Wissensstand entsprechen, und verändern Sie sie geringfügig.

5. Ein-/Ausgabebefehle

5.1 Dateneingabe

FORM INPUT { F } — **Formatierte String-Eingabe**

```
FORM INPUT Anz,Var$
```

Eine Kombination aus INPUT- und INPUT$-Befehl ist der FORM INPUT-Befehl. Sie können damit eine maximale Länge des einzugebenden Textes vorherbestimmen. Die größtmögliche Länge des Strings liegt bei 255 Zeichen. Wird vom Anwender die vorgesehene String-Länge erreicht, bleibt der Cursor an der letzten Stelle stehen. Die Eingabe kann also nur durch <Return> abgeschlossen werden. Veränderungen am Text sind auf die gleiche Art und Weise möglich, wie unter INPUT beschrieben. Das gleiche gilt für die Eingabe von Sonderzeichen. Die Angabe einer Variablenliste oder eines vorangestellten Textes ist hier allerdings nicht möglich. Es wird generell der Cursor auf die erste Stelle des Eingabebereichs gesetzt. Beispiel:

```
FORM INPUT 32,Textvar$
```

Positionieren können Sie den Eingabebereich übrigens, wie auch bei INPUT und LINE INPUT, indem Sie mit

```
PRINT AT(XP,YP);
```

den Cursor an die Stelle bringen, an der das erste Zeichen der Eingabe erscheinen soll. Durch das dem PRINT-Befehl nachgestellte Semikolon wird der Wagenrücklauf (CR = Carriage Return) und der Zeilenvorschub (LF = Line Feed) unterdrückt. Das nächste Zeichen einer Ausgabe wird dann an die Position direkt hinter dem zuletzt ausgegebenen Zeichen gesetzt.

FORM INPUT AS — Formatierte String-Eingabe m. Vorgabe

```
{ F AS }
FORM INPUT Anz AS Var$
```

Hier gilt grundsätzlich das gleiche wie bei FORM INPUT. Nur daß hier der Inhalt einer String-Variablen auf dem Bildschirm ausgegeben und die nachträgliche Edition des darin enthaltenen Strings (oder eines Teils davon) ermöglicht wird. "Anz" enthält die Anzahl der Zeichen ab Anfang des Vorgabe-Strings, die zur Edition ausgegeben werden sollen. Nach erneuter Eingabe und Bestätigung durch <Return> wird der neu entstandene String in die Variable übernommen. Der vorherige Variableninhalt wird komplett ersetzt. Wird mit 'Anz' die Länge der angegebenen Variablen nicht überschritten, muß vor der Edition anhand der <Delete>-Taste Platz zur Eingabe geschaffen werden. Bei Angabe einer Leervariablen erfüllt dieser Befehl dieselbe Funktion wie FORM INPUT. Beispiel: (in Verbindung mit MID$()= und MID$)

```
A$="String vor der Eingabe!"           ! Beliebiger String
B$=Mid$(A$,8,3)                        ! 3 Zeichen austrennen
Print At(10,10);"=> ";A$;" => ";       ! Eingabe positionieren
Form Input 7 As B$                     ! 8 Zeichen zur Eingabe
Mid$(A$,8,Len(B$))=B$                  ! Eingabe einfügen
Print At(10,10);"=> ";A$;Spc(15)       ! String ausgeben
```

INKEY$ — Einzelzeichen von Tastatur holen

```
Zeichen$=INKEY$ => Zeichen-Zuweisung
[IF/WHILE/UNTIL] LEN(INKEY$) => Bedingung: Inkey$ >""
[IF/WHILE/UNTIL] INKEY$="Z" => Bedingung: "Z" gedrückt
```

Mit INKEY$ können nicht nur die normalen Tastenbelegungen erfragt werden, sondern auch alle Sondertasten (z.B. F1 - F10 und Cursor-Block). Zusätzlich werden fast alle Kombinationen von <Control>, <Shift> oder <Alternate> mit anderen Tasten re-

gistriert. So lassen sich die Funktionstasten in Kombination mit der <Shift>-Taste doppelt belegen. Die Funktion schaltet intern auf einen anderen Modus um, sobald Sondertasten bzw. ihnen entsprechende Tastenkombinationen gedrückt werden. Im Normalfall wird ein Ein-Zeichen-String zurückgegeben, der das der gedrückten Taste entsprechende ASCII-Zeichen beinhaltet. Bei Sondertasten wird dagegen ein Mehr-Byte-String zurückgegeben, der die sogenannte CSI-Sequenz der Taste enthält. Diese wird durch ein CHR$(155) eingeleitet, gefolgt von ein bis drei weiteren Zeichen. Wird im Moment der INKEY$-Abfrage keine Taste gedrückt, wird ein Leer-String ("") geliefert. Beispiel:

```
PRINT "Testen Sie beliebige Tasten und Tastenkombinationen"
PRINT "               (Abbruch durch <Esc>)"
DO                                  ! Große Schleife
  REPEAT                            ! Eingabeschleife
    key$=INKEY$                     ! Abfrage
  UNTIL key$>""                     ! bis Taste gedrückt ist
  PRINT AT(10,10);"                                   "
  PRINT AT(10,10);                  ! Ausgabe positionieren
  IF LEN(key$)=1                    ! Ein-Byte-Code
    PRINT "ASCII :  ";
    PRINT ASC(key$)
  ELSE                              ! Mehr-Byte-Code
    PRINT "CSI-Sequenz :   ";
    FOR i%=1 TO LEN(key$)
      PRINT ASC(MID$(key$,i%));" ";
    NEXT i%
  ENDIF
  EXIT IF key$=CHR$(27)             ! Abbruch bei <Esc>
LOOP                                ! Schleifenende
```

Da gibt es allerdings ein Problem. Der Tastatur-Puffer hat die - manchmal unangenehme - Eigenart, sich die Tasten, die gedrückt wurden, zu merken. Und zwar alle! Das führt dazu, daß - wenn man ganz kontrolliert nur einen Tastendruck zulassen will - über INKEY$ alle Zeichen ausgegeben werden, die im Tastaturspeicher evtl. durch einen Dauertastendruck gespeichert wurden. Dieses Problem läßt sich beseitigen, indem man nach der INKEY$-Abfrage eine kleine Schleife einbaut, die den Tastaturpuffer leert.

```
Repeat
Until Inkey$=""
```

Um eine Einzeltastenabfrage durch INKEY$ zu realisieren (z.B. in einem alphanumerisch indizierten Menü), kennt man aus anderen BASICs die Variante:

```
100 A$=Inkey$:If A$="" Then Goto 100
```

Diese Konstruktion wird in GFA-BASIC ersetzt durch:

```
Repeat
Until Len(Inkey$)
```

oder

```
Repeat
Until Inkey$<>""
```

Sie hat die gleiche Aufgabe, nämlich darauf zu warten, daß irgendeine Taste auf dem Keyboard gedrückt wird. Um eine bestimmte Taste zu erfragen, gibt es in anderen BASIC-Dialekten die altbekannte Möglichkeit:

```
100 A$=Inkey$:If A$<>"b" Then Goto 100
```

Das gleiche in GFA-BASIC:

```
Repeat Until Inkey$="b"
```

Damit kann also kontolliert werden, ob eine bestimmte Taste (hier: b) gedrückt wurde. Wird eine andere Taste gedrückt, bleibt das Programm in der Warteschleife.

INPUT { INP } — Dateneingabe

```
INPUT ["Text";,] Var1 [,Var2,...]
INPUT #Kanal,Var1 [,Var2,...]
```

Dieses ist der noch am meisten verwendete Befehl zur Eingabe von Werten oder Strings an das Programm. Im Anschluß an die-

sen Befehl kann eine Text-Konstante angegeben werden, die vor der ersten Eingabestelle erscheinen soll.

Nach einem Komma oder Semikolon wird dann die Werte- oder Textvariable angegeben, die die eingegebenen Daten aufzunehmen hat. Der Unterschied zwischen der Angabe eines Kommas oder eines Semikolons ist der, daß bei einem Semikolon nach dem Befehl bzw. nach dem eingefügten Text ein Leer- und ein Fragezeichen erscheint, während beim Komma direkt nach dem Befehl oder Text der Cursor erscheint und dort die Eingabe beginnen kann.

Sollen mehrere Werte oder Strings über einen Input-Befehl eingegeben werden, können Sie durch Kommas getrennt eine Liste der dafür vorgesehenen Variablen anfügen. Der Befehl ist dann erst abgeschlossen, wenn für jede angegebene Variable die entsprechenden Daten eingegeben wurden. Dabei ist es sogar möglich, verschiedene Variablentypen zu verwenden. Sie können somit also mit einem Input-Befehl gleichzeitig Werte und Text erfragen. Weisen Sie jedoch dem angegebenen Variablentyp nicht die entsprechenden Daten zu (bei numerischen Variablen Text oder umgekehrt), wird ein akustisches Signal ausgegeben (außerdem blinkt der Bildschirm kurz auf), und die Eingabe kann wiederholt werden.

Jede einzelne Eingabe kann bei Mehrfach-INPUT entweder durch <Return> abgeschlossen werden, oder Sie können die Daten innerhalb einer Zeile jeweils durch ein Komma voneinander trennen. Bei Trennung durch <Return> wird dann allerdings der Cursor an den Anfang der nächsten Bildschirmzeile gesetzt und dort auf die nächste Eingabe gewartet.

Möchten Sie Kommas in der Eingabezeile verwenden, ohne daß dadurch zur nächsten Variablen weitergeschaltet wird, erreichen Sie dies, indem Sie die betreffende Antwort in Anführungsstriche setzen. Bei Verwendung einer einzelnen String-Aufnahmevariablen und Verwendung von Kommas, ohne daß die Antwort in Anführungsstriche gesetzt wurde, wird die Antwort

bei dem ersten auftretenden Komma abgeschnitten und nur dieser vordere String-Teil der Variablen zugeordnet. Beispiel:

```
Print At(10,10);
Input "Eingabe Wert,String,Wert: ",A%,B$,C%
Print A%,B$,C%
```

Bei String-Eingaben sind Strings mit einer Länge von bis zu 255 Zeichen möglich. Während der Eingabe können Sie die schon eingegebenen Daten auf verschiedene Weise korrigieren:

<Backspace>	Zeichen links vom Cursor löschen
<Delete>	Zeichen unter dem Cursor löschen
<Pfeil-links>	Cursor um eine Stelle nach links
<Pfeil-rechts>	Cursor um eine Stelle nach rechts (sofern er nicht am Zeilenende steht)
<Pfeil-hoch>	Cursor zum Zeilenanfang
<Pfeil-runter>	Cursor zum Zeilenende

Wollen Sie bei der Eingabe Sonderzeichen verwenden, gibt es dazu zwei Möglichkeiten:

1. <Alternate> und andere <Taste>
2. <Amiga> und andere <Taste>

Wie bei PRINT gibt es auch hier die Kanal-Variante. Mit INPUT # und der daran anschließenden Nummer des gewünschten Datenkanals können auch Daten aus Diskettendateien bzw. über die Schnittstellen (siehe OPEN) bezogen werden. Beispiel:

```
Open "I",#1,"DATEI.DAT"
Input #1,A$,B$,C$
```

oder:

```
Open "",#1,"CON:"
Input #1,A$,B$,C$
```

Bei diesen Beispielen muß allerdings gewährleistet sein, daß die aus der Datei gelesenen Daten auch zu den vorgegebenen Variablentypen passen.

INPUT$() Zeichenketteneingabe

```
A$=INPUT$(Anz)
A$=INPUT$(Anz,#Kanal)
```

Wollen Sie, daß Strings "verdeckt" eingegeben werden, können Sie das mit dieser Eingabe-Funktion erreichen. Anders als bei allen anderen Input-Arten wird hier die aufnehmende Variable nicht dem Befehl nachgestellt, sondern der String wird einer Variablen zugewiesen (A$=INPUT$(10)), direkt nach Abschluß der Eingabe ausgegeben (PRINT INPUT$(10)) oder in eine Abfrage eingebunden (IF INPUT$(4)="xxxx").

Der Funktion wird in Klammern eine Zahl übergeben, die angibt, wie viele Zeichen maximal eingegeben werden können. Wird diese Zeichenanzahl erreicht, wird das Programm, ohne auf die Betätigung der <Return>-Taste zu warten, fortgesetzt. Der eingegebene Text ist während der Eingabe nicht zu sehen. Korrekturen an der Eingabe sind auf dieselbe Art wie bei INPUT möglich.

Auch hier gibt es die Kanal-Variante (A$=INPUT$(10,#1)). Dieser Befehl eignet sich besonders dazu, eine bestimmte Datenmenge aus einer Diskettendatei zu lesen. Um zum Beispiel eine ganze Datei in einen String einzulesen (vorausgesetzt, sie ist nicht länger als die maximale String-Länge von 32767 Byte), können Sie folgendes Mini-Programm verwenden (siehe auch BGET# bzw. LINE INPUT#):

```
Open "I",#1,"Datei.Dat"
A$=Input$(Min(32767,Lof(#1)),#1)
Close #1
Print A$
```

Sie müssen hierbei die Länge der Datei nicht kennen, da sie durch LOF() ermittelt wird. Um die maximal mögliche String-Länge (32767 Zeichen) nicht zu überschreiten, wird die einzulesende Zeichenanzahl durch MIN() auf diese maximale Anzahl begrenzt.

LINE INPUT { LI } Zeichenketteneingabe

```
LINE INPUT ["Text";,] Var$ [Var2$,...]
LINE INPUT #Kanal, Var$ [Var2$,...]
```

Der LINE INPUT-Befehl ist ein direkter Nachkomme des normalen INPUT-Befehls. Sie unterscheiden sich dadurch, daß LINE INPUT ausschließlich für die Texteingabe zu verwenden ist, und daß bei Angabe einer Variablenliste die Eingaben alle einzeln mit <Return> abgeschlossen werden müssen. Während bei INPUT durch die Eingabe eines Kommas kenntlich gemacht wird, daß nun die nächste Variable in der Liste bedient wird, wird hier das Komma als Textzeichen innerhalb des eingegebenen Strings anerkannt. Genauso wie bei INPUT kann auch hier ein Text-String übergeben werden, der dann vor dem ersten einzugebenden Zeichen erscheint.Beispiel:

```
Line Input "Bitte Text eingeben: ",Aa$,Bb$,Cc$
```

Durch Verwendung eines Kommas oder Semikolons zwischen dem Befehl (bzw. Text) und der Variablen (bzw. Variablenliste) kann auch hier bestimmt werden, ob ein Leerzeichen und ein Fragezeichen vor der ersten Eingabestelle erscheinen soll oder nicht. Editionsmöglichkeiten wie bei INPUT.

Die zweite Syntaxform liest die Daten aus der in Kanal angegebenen Datei oder Schnittstelle (siehe OPEN). Hier erklärt sich auch, warum dieser Befehlsname mit LINE beginnt. Befinden sich nämlich innerhalb des eingelesenen Textes keine CRs oder LFs, bzw. zwischen einem CR/LF und dem nächsten liegen mehr als 254 Zeichen, wird der Text nach maximal 254 Zeichen

automatisch abgeschnitten und der betreffenden Variablen zugeordnet. Texteingaben mit mehr als 254 Zeichen (254+CR+LF = 256 = max. Zeilenlänge) sind nicht möglich.

Sind im Text Zeichen mit einem ASCII < 32 enthalten, die keine Zeilentrennzeichen (CR/LF) sind, sind Fehleingaben möglich.

Etwas verwirrend ist, daß in Fällen, in denen nur ein Carriage Return zur Zeilentrennung verwendet wurde, dies zwar insoweit erkannt wird, daß nur der vordere Teil des Textes bis zum ersten CR an die Variable übergeben wird, jedoch der File-Pointer um die gesamte Anzahl der gelesenen Zeichen weitergesetzt wird.

Folgerung: LINE INPUT# ist nur für ASCII-Text (ASCII > 31) - mit CR+LF als Zeilentrennung - korrekt einsetzbar.

5.2 Datenausgabe

PRINT { ? oder P } **Daten ausgeben**

```
PRINT [AT(S,Z)] [,']"Text"[[;,']Var[;,']Expr...;]
PRINT [#Kanal,[;]] "Text"[[;,']Var[;,']Expr...;]
```

Mit PRINT läßt sich fast alles auf den Bildschirm schreiben. Er ist einer der Tausendsassa-Befehle die man in BASIC kennt. Um seinen Variantenreichtum ausloten zu können, wird es Ihnen nicht erspart bleiben, ihn in verschiedenen Fällen bei der Arbeit zu beobachten. Probieren Sie ihn also fleißig aus. Beispiel:

```
A%=1
A$="- Demo"
Print At(1,12);
Print Chr$(27);"p";" PRINT"'A$,A%,"Taste: ";
Print "!";Chr$(27);"q";At(1,13);
Print String$(40,Left$(A$))
```

In diesen bunt zusammengesetzten PRINT-Befehlen haben die einzelnen Komponenten folgende Bedeutung:

```
At(XP,YP)
```

Mit diesem wahlfreien Zusatz können Sie bestimmen, an welcher Cursor-Position die Ausgabe erfolgen soll. Das Ausgabefenster besteht, von der Home-Position aus gezählt, aus 77 Spalten und 28 Zeilen (XP= 1-77/YP= 1 - 28).

Dasselbe wird auch durch LOCATE erreicht (siehe dort). AT(XP,YP) kann - wie im Beispiel gezeigt - auch innerhalb der Ausgabezeile eingesetzt werden, wodurch mit einem PRINT verschiedene Bildschirmpositionen bestimmbar sind.

; (Semikolon)

Dieses Zeichen dient dazu, verschiedene Ausdrücke miteinander zu verbinden. Der nach diesem Zeichen stehende Ausdruck wird dann direkt an den vorhergehenden angeschlossen. Wenn das Semikolon als Schlußzeichen eingesetzt wird, wird auch der erste Ausdruck des nächsten PRINT-Befehls an den zuletzt ausgegebenen PRINT-Ausdruck angehängt, da Carriage Return und Line Feed (CR/LF = Wagenrücklauf/neue Zeile) in diesem Fall unterdrückt werden.

, (Komma)

Wird das Komma eingesetzt, dann wird der folgende Ausdruck an den nächsten von 5 Tabulatorpunkten hinter der letzten Ausgabe gesetzt. Diese Punkte haben die X-Positionen 1,17,33,49,65.

' (Apostroph)

Ein Apostroph steht als Leerzeichen. Es werden also in der Bildschirmausgabe überall dort Leerzeichen gesetzt, wo sie durch dieses Zeichen vorgegeben wurden.

Außerdem wurden String- und numerische Funktionen in die Zeilen eingebaut, um zu zeigen, daß Funktionen problemlos di-

rekt in die PRINT-Ausgabe integriert werden können. Beachten Sie dazu die jeweilige Funktionsbeschreibung unter CHR$(), STRING$(), LEFT$() etc.

Ein direkt angegebener Text ist vor dem ersten Textzeichen mit Anführungszeichen zu versehen (Print "text), da sonst der Text ggf. als Variable ohne Inhalt interpretiert wird. Folgt auf den Text nichts mehr, setzt der Interpreter die Ausführungszeichen am Ende selbständig.

Ein einzelner PRINT-Befehl ohne nachgestellte Ausdrücke bewirkt den Ausdruck einer Leerzeile (CR/LF). Eine Variante des PRINT-Befehls ist PRINT #. In diesem Fall folgt auf das Nummernzeichen eine Zahl, die den anzusprechenden Datenkanal angibt. Wenn Sie also in eine geöffnete Diskettendatei etwas hineinschreiben möchten, ist das hiermit möglich (z.B. Print#1;"BASIC").

PRINT USING { P USING } — Daten formatiert ausgeben

```
PRINT USING "format",Expr [,Var,...][;]
PRINT USING Format$,Expr [,Var,...][;]
PRINT #Kanal,USING "format",Expr [,Var,...][;]
```

Durch diesen Befehl kann ein Ausgabeformat für Werte und Strings bestimmt werden. Ihm folgt als erstes ein String oder eine String-Variable, in der das gewünschte Format angegeben wird. Als nächstes werden nach einem Komma die Ausdrücke oder Variablen übergeben, die diesem Format entsprechen sollen. Werden mehrere Ausdrücke angegeben, müssen diese durch weitere Kommas getrennt werden. Als Formatierungszeichen sind vorgesehen:

\# Platzhalter für eine Ziffer:

```
Print Using "######",Int(31421/3)
```

Aursgabe: 10473

. Position des Dezimalpunktes:

```
Print Using "######.####",31421/4
```

Ausgabe: 7855.2500

\+ Ausgabe auch des positiven Vorzeichens:

```
Print Using "+######.####",31421/4
```

Ausgabe: + 7855.2500

\- Platzhalter für negatives Vorzeichen:

```
Print Using "-#####.######",31421/-4
```

Ausgabe: - 7855.250000

* Füllzeichen für alle angegebenen Vorkommastellen, die nicht von dem auszugebenden Wert belegt werden. Sonst wie #.

```
Print Using "*######.####",31421/1.4
```

Ausgabe:**22443.5714

Hinter dem Dezimalpunkt verwendet, werden so viele * ausgegeben, wie angegeben sind, und die auszugebende Zahl wird real auf die gewünschte Stelle gerundet.

```
Print Using "*######.#****",31421/1.4
```

Ausgabe:**22443.6****

$ Voranstellung eines $:

```
Print Using "$######.##",31421/1.4
```

Ausgabe: $22443.57

, Einfügen eines Kommas (Tausendertrennung):

```
Print Using "##,###,###.###",3142*2781.71
```

Ausgabe: 8,740,132.820

^^^^ Ausgabe im Exponentialformat. Führende # stehen hier für die Stellen des Basis-Anteils und ^ für die Exponentenstellen (E+xxxx). Überflüssige Basis-Stellen werden mit 0 gefüllt. Der Exponent wird den Vorkommastellen angepaßt.

```
Print Using "#.########^^^^",13711*64
```

Ausgabe:8.77504000E+05

! Das erste Zeichen eines Strings wird ausgegeben:

```
Print Using "!sing !n !FA","Uhu","igitt","Gaga"
```

Ausgabe:Using in GFA

& Gesamt-String wird ausgegeben:

```
Print Using "&hausen","Enten"
```

Ausgabe: Entenhausen

\..\ Ausgabe von so vielen Zeichen des Strings, wie Länge von \..\ (inkl. Backslashs):

```
Print Using "\..\ingen","Hattu Möhren?"
```

Ausgabe: Hattingen

_ (Tiefstrich) Interpretiert das hierauf folgende Using-Formatzeichen nicht als solches, sondern gibt es als ASCII-Zeichen aus:

```
Print Using "Channel _### _\ &",44,"XYZ"
```

Ausgabe: Channel #44 \ XYZ

Die Formatvorgabe und die String- und/oder Werte- und/oder Ausdrucksliste kann in beliebiger Reihenfolge gegliedert sein, solange die Parametertypen in ihrer Reihenfolge der Formatvorgabe entsprechen. Wird die Reihenfolge nicht korrekt eingehalten bzw. trifft der Interpreter auf unlogische Formate, wird ein Fragezeichen an der betreffenden Stelle ausgegeben. Bei numerischen Formaten wird dem ausgegebenen Wert ein Prozent-

zeichen vorangestellt, wenn die Stellenanzahl des Wertes das dafür vorgesehene Format überschreitet. Werden außerdem bei numerischen Werten im Format weniger Nachkommastellen angegeben als vorhanden sind, werden die übrigen Nachkommastellen integriert.

Außerdem haben Sie die Möglichkeit, zwischen Dezimalpunkt und -komma zu wählen (siehe MODE). Durch Nachstellen eines Semikolons (Print Using "....",Expr,Liste....;) kann - wie bei PRINT - die Ausgabe von CR/LF unterdrückt werden.

Mit Angabe eines Datenkanals (Print #1,Using...) können die Ausgaben auch auf diesen Kanal umgeleitet werden. Die oben angeführten Syntaxregeln sind auch dann gültig.

WRITE { WR } — Daten ausgeben

```
WRITE [#Kanal,] ["Text" [,Var,Expr;...]]
```

Das ist ein Ausgabe-Befehl, der in erster Linie zur Datenspeicherung in sequentiellen Dateien gedacht ist. Dieser Befehl kann aber auch zur Text- bzw. Datenausgabe auf dem Bildschirm verwendet werden. Er ist dem PRINT-Befehl sehr ähnlich, hat jedoch einen anderen syntaktischen Aufbau. Außerdem werden hier die Anführungszeichen eines übergebenen Strings (Ausdruck oder Variable) sowie die Kommas, mit denen hier die einzelnen Ausdrücke voneinander getrennt werden, ebenfalls ausgegeben.

Seinen eigentlichen Sinn zeigt dieser Befehl jedoch erst, wenn mit einem INPUT#-Befehl mehrere Werte oder Strings gleichzeitig aus einer Datei eingelesen werden sollen. Dazu müssen die Einzeldaten durch Kommata voneinander getrennt sein. Da WRITE# Kommata an den entsprechenden Platz in der Datei schreibt, können die Daten beim Einlesen mit INPUT# unterschieden und den dabei angegebenen Variablen zugeordnet werden. Beispiel:

```
Open "O",#1,"Friends"                  ! Datei zur Ausgabe öffnen
Restore N_amen                         ! Data-Zeiger setzen
For I%=1 To 4                          ! 4 Zeilen
  Read Name$,Beruf$,Telefon$           ! Je 3 Datas
  Write #1,Name$,Beruf$,Telefon$       ! in die Datei schreiben
Next I%                                ! Nächste Zeile
Close #1                               ! Datei schließen
N_amen:
Data " Elizabeth "," Königin          "," London/112233          "
Data " Kashogghi "," Milliardär       "," Riad/1.000.000.000     "
Data " Boris     "," The lost Winner  "," Leimen/666666          "
Data " Yeti      "," Schneemensch     "," Himalaya/XY-ungelöst   "
Open "I",#1,"Friends"                  ! Datei zum Einlesen öffnen
Print "Meine besten Freunde:";Chr$(13);Chr$(10)     ! Bla...
Print "Datei-Inhalt ohne Format:";Chr$(13);Chr$(10) ! ...bla
A$=Input$(Lof(#1),#1)                  ! Kompletten Datei-Inhalt lesen
Print A$                               ! Unformatiert ausgeben
Seek #1,0                              ! File-Pointer wieder auf Anfang
Print "Mit WRITE ausgegeben:";Chr$(13);Chr$(10)
For I%=1 To 4                          ! 4 Zeilen
  Input #1,N.ame$,B.eruf$,T.elefon$ ! Je 3 Ausdrücke
  Write N.ame$,B.eruf$,T.elefon$       ! mit WRITE ausgeben
Next I%
Seek #1,0                              ! File-Pointer wieder auf Anfang
Print Chr$(10);"Mit PRINT ausgegeben:";Chr$(13);Chr$(10)
For I%=1 To 4                          ! 4 Zeilen
  Input #1,N.ame$,B.eruf$,T.elefon$ ! Je 3 Ausdrücke
  Print N.ame$,B.eruf$,T.elefon$       ! mit PRINT ausgeben
Next I%
Close #1                               ! Datei schließen
```

5.3 Bildschirmoperationen

HTAB { HT } — **Aktuelle Cursor-Spalte bestimmen**

```
HTAB Spalte
```

Ein Befehl, der vor allem der Kompatibilität zu anderen BASIC-Dialekten und der Zusammenarbeit mit CRSCOL dient.

LOCATE { LOCAT } — Cursor positionieren

```
LOCATE S,Z
```

Positioniert den Zeichen-Cursor auf Spalte "S" und Zeile "Z", analog zu PRINT AT(S,Z). Weiteres siehe dort.

POS() — Cursor-Spalte ermitteln

```
Var=POS(Dummy)
```

POS liefert Ihnen die Zahl der seit dem letzten Wagenrücklauf (CR) am Bildschirm ausgegebenen Zeichen AND 255. Es muß ein in Klammern nachgestelltes numerisches Scheinargument angegeben werden. Diese Zahl kann beliebig gewählt werden und hat keinen weiteren Einfluß auf die Funktion.

Da eine Bildschirmzeile maximal 80 Zeichen aufnehmen kann, muß der von POS zurückgelieferte Wert nicht unbedingt mit der tatsächlichen Spaltenposition des Cursors übereinstimmen. Geben Sie zum Beispiel eine 150 Zeichen lange Zeichenkette aus, so liefert POS den Wert 150, der Cursor steht jedoch in Spalte 70.

Ist in der letzten Ausgabe eines der Steuerzeichen CHR$(8) (Backspace) oder CHR$(13) (Carriage Return) enthalten gewesen, haben diese Einfluß auf die POS()-Position:

```
Backspace        (BS=Chr$(8))  Vermindert POS() um 1
Carriage Return  (CR=Chr$(13)) Setzt POS() auf Null
```

SPC() Leerzeichen ausgeben

```
PRINT SPC(Anz)
PRINT [Ausdrücke;Werte;etc.;] SPC(Anz) [;etc.]
```

SPC ist ein Befehl, der nur im Zusammenhang mit PRINT verwendet werden kann. Der in Klammern angegebene Ausdruck Anz steht für die Anzahl an Leerzeichen (SPaCe = 0 - 255), die an der aktuellen Cursor-Position ausgegeben werden sollen. Beispiel:

```
Print "==>";Spc(20);"Ende"
     Ausgabe : ==>          Ende
```

Nicht möglich ist:

```
A$="==>"+Spc(20)+"Ende"
```

Es wird ein Syntaxfehler angezeigt. Für solche Konstrukte eignet sich SPACE$(20).

TAB() Tabulator setzen

```
TAB(Position)
PRINT [Ausdrücke;Werte;etc.;] TAB(Anz) [;etc.]
```

Es kann eine Tabulatorposition bestimmt werden, an welcher der Cursor dann positioniert wird. Diese Position kann im Bereich von 0 bis 255 liegen. Größere Werte werden mit MOD 256 auf diesen Bereich zurückgerechnet. Befindet sich der Cursor in einer Zeile hinter der zuletzt angegebenen Tabulatorposition und der Wert einer sich anschließenden TAB-Anweisung liegt zwischen 256 und der aktuellen Cursor-Position, so wird dieser Tabulator in derselben Zeile ausgeführt. Ist der TAB-Wert kleiner als die aktuelle Cursor-Position, wird die Anweisung in der

nächsten Zeile ausgeführt. TAB ist ebenso wie SPC nur in Verbindung mit PRINT ausführbar. String-Konstrukte mit TAB sind nicht möglich. Beispiel:

```
For J%=0 To 11                  ! 12 mal
  Restore T_ext                 ! Data-Zeiger setzen
  For I%=1 To 2                 ! 4 Datas...
    Read A$,A%                  ! ...lesen...
    Print Tab(J%*6);A$''''''A%  ! ... und ausgeben
  Next I%                       ! Nächstes Data
Next J%                         ! Nächste Position
T_ext:
Data GFA-,1,BASIC,2
```

VTAB { VT} — Aktuelle Cursor-Zeile bestimmen

```
VTAB Zeile
```

VTAB und HTAB (siehe dort) haben im Vergleich zu LOCATE und PRINT AT den Vorteil, daß sich mit ihnen der Cursor für Zeile und Spalte getrennt positionieren läßt.

5.4 Diskettenoperationen

Bei allen Diskettenoperationen, denen ein Dateiname zu übergeben ist, besteht die Möglichkeit, einen Suchpfad zu definieren, über den der Dateizugriff ausgeführt werden soll. Dieser setzt sich (entsprechend den Konventionen von AmigaDOS) aus drei Komponenten zusammen:

```
[Diskettenname | Laufwerk ]:[Verzeichnisname/.../Verzeichnisname/]
Dateiname
```

Diskettenname | Laufwerk:

Ganz am Anfang des Pfads steht entweder der Name der anzusprechenden Diskette oder die Bezeichnung des gewünschten

Laufwerks, beides gefolgt von einem Doppelpunkt. Als Laufwerksbezeichnung sind die folgenden Namen zulässig:

```
DF0:, DF1:,DF2:,DF3: für die (maximal) vier Diskettenlaufwerke
DH0:, DH1:, ...: für die Partitions auf einer Amiga-seitigen Harddisk
JH0:, JH1:, ...: für die Amiga-Partitions auf einer PC-seitigen Hard
                 disk
RAM: für evtl. vorhandene RAM-Disk
```

Die Angabe des Laufwerks kann auch entfallen. In diesem Fall wird dann auf dem aktuellen Laufwerk gesucht.

Verzeichnisname:

An zweiter Stelle folgt der Name des Verzeichnisses (Ordners), in dem die Datei abgelegt ist. Verzeichnisse dürfen beliebig verschachtelt werden, (mehr als drei Ebenen sind jedoch aus Gründen der Übersichtlichkeit nicht ratsam). Die einzelnen Namen müssen in diesem Fall durch Schrägstriche (/) voneinander getrennt werden. Jeder Name darf maximal 30 Zeichen enthalten.

Dateiname:

Der Dateiname (von den Verzeichnisnamen durch einen Schrägstrich getrennt) darf ebenfalls bis zu 30 Zeichen lang sein. Beispiele:

```
DF1:UTILITY/OUTPUTS/DRUCKE.WAS
```

sucht im Laufwerk DF1 im Unterordner OUTPUTS des Ordners UTILITY nach der Datei DRUCKE.WAS.

```
PROGRAMME:GFA-BASIC/GRAFIK.LST
```

sucht auf der Diskette PROGRAMME im Ordner GFA-BASIC nach der Datei GRAFIK.LST.

```
dh0:BEISPIELE
```

sucht auf der Amiga-seitigen Harddisk-Partition 0 nach der Datei BEISPIELE. Der Pfadname kann unter GFA-BASIC in

den meisten Fällen als String-Ausdruck, als String-Variable oder als Kombination von beidem übergeben werden.

Im folgenden wird Ihnen öfter der Begriff "#Kanal" begegnen. Damit ist bei Datei-relativen Diskettenoperationen der Identi-fikator (0 - 99) der jeweils angesprochenen Datei gemeint.

BLOAD { BL } — Datei in Speicherbereich laden

```
BLOAD "Dateiname" [,Start]
```

Mit BLOAD kann eine beliebige Datei komplett vom Fest-speicher (Diskette/Hard-Disk/RAM-Disk) an eine beliebige RAM-Adresse geladen werden.

In "Dateiname" wird die Dateibezeichnung (ggf. inkl. Pfad) übergeben, und durch den Parameter "Start" wird die Adresse angegeben, ab welcher die Daten abgelegt werden sollen. Wird "Start" ausgelassen, wird jene Adresse als Ziel verwendet, die beim letzten BSAVE-Aufruf als Quelle gedient hat. Beide Parameter können auch in Variablen übergeben werden.

Vor einem BLOAD-Aufruf sollten Sie sicherstellen, daß die zu ladende Datenmenge auch tatsächlich ab der angegebenen Speicherstelle untergebracht werden kann, ohne in wichtige Bereiche hineinzuschreiben. In den meisten Fällen wird dies durch das Einrichten eines ausreichend großen Puffers in Form einer Stringvariablen erledigt.

BSAVE { BS } — Speicherbereich auf Disk speichern

```
BSAVE "Dateiname",Start,Anz
```

Das ist ein sehr komfortabler Befehl, auf den man sicher immer wieder zurückkommen wird. Er bietet die Möglichkeit, eine beliebige Anzahl von Bytes als gesamten Block (BSAVE = Block-SAVE) auf Diskette zu speichern. Es muß der Name der Datei angegeben werden, die diesen Block aufnehmen soll. Nach einem Komma folgt die Adresse, in welcher das erste Byte des Blocks steht, und abschließend wieder nach einem Komma die Anzahl (Anz) der Bytes, die gespeichert werden sollen.

CHAIN { CHAI } — Programm laden (Autostart)

```
CHAIN "Programmname"
```

Dieser Befehl ist eigentlich identisch mit LOAD. Allerdings wird das hiermit geladene Programm nach dem Laden automatisch gestartet. Da bei diesem Programmstart - wie sonst auch - alle Variablen und Felder gelöscht werden, können diese zwischen den "gechainten" Programmen nicht ausgetauscht werden. Hier hat man nun folgende Möglichkeit:

Sie können alle wichtigen Daten, die von dem aufrufenden Programm übergeben werden sollen, in eine Datei schreiben (siehe PRINT#/WRITE#), diese am Programmanfang des CHAIN-Programms einlesen (siehe INPUT#) und die Puffer-Datei dann wieder löschen. Der Vorteil ist hier, daß die übergebene Datenmenge nur durch den freien Disk-Speicherplatz begrenzt ist. Beispiel:

Programm 1:

```
A%=1025                  ! Beliebige Variable
B=399.22                 !     "        "
```

```
C$="BASIC"                 !     "         "
Open "O",#1,"Vars.Dat"     ! Puffer-Datei öffnen
Write #1,A%,B,C$           ! Daten übergeben
Close #1                   ! Datei schließen
Chain "Program2.GFA"       ! Programm 2 aufrufen
Programm 2 (Program.GFA):
Open "I",#1,"Vars.Dat"     ! Puffer-Datei öffnen
Input #1,A%,B,C$           ! Daten einlesen
Close #1                   ! Datei schließen
Kill "Vars.Dat"            ! Puffer-Datei löschen
Print "Variablen aus Programm 1 : ";A%''B''C$
```

Im Interpreter-Betrieb wird durch CHAIN auch der BASIC-Arbeitsspeicher samt Inhalt (aufrufendes Programm) vorher gelöscht. Wenn Sie kein PSAVE-geschütztes Programm aufgerufen haben, finden Sie nach Programmende das nachgeladene Programm auch im Editor.

CHDIR { CHD } — Ordner wechseln

```
CHDIR "Ordner"
```

Hiermit wird das aktuelle Verzeichnis (Ordner) neu festgelegt. Dieses Verzeichnis gilt im weiteren Programm (bis zur nächsten Änderung) als voreingestellt, d.h. bei Diskettenzugriffen wird immer auf dieses Verzeichnis zugegriffen. Dem Befehl wird einfach der Zugriffspfad übergeben, über den das Verzeichnis zu erreichen ist (Pfadstruktur siehe Kapitelanfang). Diese Bezeichnung kann wieder entweder in einer Variablen enthalten sein (Chdir Ordner$) oder als Textkonstante übergeben werden (Chdir "UTILITY").

Besteht die Pfadangabe nur aus einem Doppelpunkt, greift CHDIR danach nicht auf einen Ordner, sondern auf das Hauptverzeichnis zu.

Außerdem besteht die Möglichkeit, durch einen Schrägstrich (/) in das jeweils übergeordnete Verzeichnis zu wechseln. Angenommen, das aktuelle Verzeichnis 'Ordner1' liegt im Verzeichnis 'Chef_Ordner'. Ebenfalls in diesem Über-Ordner befindet sich

noch ein weiteres Verzeichnis 'Ordner2', zu dem gewechselt werden soll. Anstatt in diesem Fall CHDIR "Chef_Ordner/Ordner2" zu schreiben, genügt ein CHDIR "/Ordner2".

DFREE() — Freien Disketten-Speicherplatz ausgeben

```
Var=DFREE(0)
```

DFREE(0) liefert den momentan freien Speicherplatz des aktuellen (über CHDIR gewählten) Laufwerks. Der Wert 0 ist ein Dummy-Wert und hat keine weitere Bedeutung.

DIR — Directory ausgeben

```
DIR ["Pfad"] [TO "Datei"]
```

Durch DIR lassen sich auf verschiedene Weise Inhaltsverzeichnisse entweder einer beliebigen Diskettenstation oder eines bestimmten Ordners ausgeben. Dabei ist es auch möglich, diese Directories in eine bestimmte Diskettendatei zu schreiben oder auf dem Drucker ausdrucken zu lassen. Zur Pfadangabe können sämtliche am Kapitelanfang beschriebenen Spezifikationen verwendet werden.Beispiele:

```
DIR "DF0:PROGRAMME"
```

gibt alle Dateien aus, die sich im Laufwerk DF0: im Verzeichnis PROGRAMME befinden.

```
DIR "DF1:" TO "PRT:"
```

gibt alle Dateien des Laufwerks DF1: auf dem Drucker aus.

DIR$() Aktuellen Zugriffspfad ermitteln

```
Var$=DIR$(0)
```

Mit dieser Funktion kann der momentan gültige (über CHDIR festgelegte) Zugriffspfad ermittelt werden. Ist kein Ordner geöffnet, wird ein Leer-String zurückgegeben.

EXIST() Existenz einer Datei prüfen

```
Var=EXIST(Dateiname)
```

Ermittelt, ob die Datei (oder auch der Ordner) "Dateiname" vorhanden ist. Es wird entweder eine 0 (= FALSE -> nicht vorhanden) oder -1 (= TRUE -> vorhanden) zurückgegeben. Die Angabe eines Suchpfades erfolgt ggf. gemäß der am Kapitelanfang beschriebenen Struktur.Beispiel:

```
If Exist("DF1:Datei.Dat")       ! Datei.Dat auf DF1?
  Open "I",#1,"DF1:Datei.Dat"   ! Ja, dann öffnen
  '...weiteres Programm
Else
  Print "Datei.Dat existiert nicht!"
Endif
```

FILES Directory (erweitert) ausgeben

```
FILES ["Pfad"] [TO "Datei"]
```

Genügen Ihnen die reinen Dateinamen nicht, die mit DIR ausgegeben werden, können Sie mittels dieses Befehls auch weitere Attribute der einzelnen Dateien erfahren. So haben Sie die Möglichkeit, zusätzlich die Größe der Dateien sowie das Datum und die Uhrzeit ihrer Erstellung in Erfahrung zu

bringen. Im übrigen wird dieser Befehl exakt genauso angewendet wie DIR (FILES z.B. listet alle Dateien der aktuellen Disk außerhalb von Ordnern sowie alle Ordner auf).

KILL { K } — Disk-Datei löschen

```
KILL "Dateiname"
```

Möchten Sie eine Datei auf dem Festspeicher (Diskette/Hard-Disk/RAM-Disk) löschen, können Sie das mit diesem Befehl tun. Dazu wird in "Dateiname" der Name der zu löschenden Datei (Pfad siehe ggf. Kapitelanfang) angegeben.

LIST { LIS } — Programm listen/speichern (ASCII)

```
LIST ["Dateiname"]
```

Möchten Sie das im Arbeitspeicher befindliche Programm auf dem Monitor auflisten oder als ASCII-File auf Diskette abspeichern, hilft Ihnen dieser Befehl. Geben Sie im Direktmodus oder als Programmzeile nur den Befehlsnamen ohne Angabe eines Dateinamens ein, wird das gesamte Programm auf dem Ausgabebildschirm ausgegeben. Das Listing kann jederzeit durch die GFA-Abbruchtastenkombination <Control><Shift><Alter-nate> unterbrochen werden. Wollen Sie das Listing als sogenanntes ASCII-File auf Diskette abspeichern, übergeben Sie dem Befehl (in Anführungsstrichen) den Namen der Datei, in welcher das Listing abgelegt werden soll. Dieser Befehl ist identisch mit der Editor-Funktion Save,A. Sie können also hiermit abgespeicherte Programme jederzeit mit der Editor-Funktion Merge in ein anderes im Arbeitsspeicher befindliches Programm einbinden. Die Anführungsstriche zum Dateinamen können vernachlässigt werden, da sie vom Interpreter - falls nicht vorhanden - selbsttätig gesetzt werden. Die Extension zum Dateinamen kann ebenfalls

vernachlässigt werden. Auch sie wird vom Interpreter gesetzt (.LST - falls nicht anders vorgegeben). Befindet sich bereits ein Programm gleichen Namens auf der Diskette, wird es automatisch auf Programmname.BAK umbenannt.

Ein ASCII-File ist eine Datei, in die jedes einzelne Text- und Steuerzeichen des zu speichernden Textes in der Reihenfolge seines Auftretens als entsprechender ASCII-Wert (0 - 255) geschrieben wird. Die meisten textverarbeitenden Programme bieten die Möglichkeit, ASCII-Files zu speichern und zu laden. D.h., daß Texte dieser Art unter den verschiedenen Programmen, Systemen und Computern austauschbar sind, soweit diese sich an den American Standard Code for Information Interchange halten (ASCII = Amerikanischer Standard-Code zur Informationsübertragung). Dateien mit anderen Formaten, wie z.B. die .GFA-Dateien werden nach einem anderen Schema kodiert. Dieser Token-Code wird vom Programmierer selbst entwickelt, um zum Beispiel eine höhere Lade- oder Speichergeschwindigkeit, einen speziellen Effekt oder eine erschwerte Lesbarkeit zu erreichen (siehe PSAVE). Diese Dateien sind dann allerdings nicht mehr mit anderen Programmen (Interpretern etc.) austauschbar. Sie sind nicht mehr kompatibel (compatibility = Vereinbarkeit/Verträglichkeit).

Zur Ausgabe des Listings auf dem Drucker haben Sie auch die Möglichkeit, in Dateiname den Drucker-Port PRT: anzugeben. Diese Variante ist dann identisch mit dem Befehl LLIST, wobei jedoch Punkt-Befehle (siehe Editor-Funktion Llist) unberücksichtigt bleiben.

LOAD { LOA } — Programm in Arbeitsspeicher laden

```
LOAD "Programmname"
```

Mit LOAD kann ein beliebiges GFA-BASIC-Programm in den Arbeitsspeicher geladen werden. Prinzipiell verhält sich dieser Befehl so, als würden Sie den Menüpunkt Load im Editormenü

anklicken oder im Editor die Taste <F1> drücken. Auch hier ist die am Kapitelanfang beschriebene Pfadstruktur zu verwenden. "Programmname" ist der Name des zu ladenden Programms. Wird dem Programmnamen keine Extension beigefügt, wird vom Interpreter selbständig .GFA angehängt, z.B.:

```
Load "Utility/Drei_d"
```

Das aktuelle Programm wird beendet und das Programm 'Drei_d' nachgeladen. Dieses muß nun durch Run im Direktmodus, <Shift><F10> oder Klick auf Run im Editormenü gestartet werden. Bei durch PSAVE gespeicherten Programmen ist dies jedoch nicht nötig, da diese automatisch gestartet werden.

MKDIR { MK } — Ordner erzeugen

```
MKDIR "Ordner"
```

Mit MKDIR können auf einer beliebigen Station im Haupt-Directory oder innerhalb eines vorhandenen Ordners weitere Ordner angelegt werden. Dabei wird ggf. wieder die am Kapitelanfang beschriebene Pfadstruktur verwendet. Als Überblick folgt eine Auflistung verschiedener Varianten dieses Befehls.

```
Mkdir "AKTE"
```

erzeugt einen neuen Ordner mit dem Namen AKTE auf der aktuellen Station entweder im Haupt-Directory oder. in dem Ordner, der gerade geöffnet ist.

```
Mkdir "DF0:FACH_A/AKTE"
```

erzeugt einen neuen Ordner mit Namen AKTE auf Station DF0: im vorhandenen Ordner FACH_A. Ist bereits ein Ordner mit derselben Pfadbezeichnung vorhanden, wird eine Fehlermeldung ausgegeben.

NAME { NA AS } — Datei umbenennen

```
NAME "Name_alt" AS "Name_neu"
```

Ermöglicht die nachträgliche Änderung eines Dateinamens. Dieser Befehl erwartet zwei Parameter. Der erste davon (Name_alt) ist der Dateiname, der verändert werden soll. Nach einem angehängten AS wird ein weiterer Name (Name_neu) übergeben, der nun an die Stelle des alten Namens tritt. Beide Namen können entweder als String-Ausdruck, als String-Variable oder als Kombination aus beidem übergeben werden. Am Kapitelanfang finden Sie die Pfadstruktur, die ggf. auch hier zu verwenden ist. Zu beachten ist bei diesem Befehl, daß eine eventuelle Laufwerksvorgabe (z.B. DF0:) bei beiden Namen identisch anzugeben ist. Ist beim alten Namen eine Diskettenstation angegeben und ist diese Station auch die aktuelle, dann kann beim neuen Namen die Spezifikation entfallen. Innerhalb einer Station kann auch ohne weiteres eine im Haupt-Directory verzeichnete Datei durch eine entsprechende Pfadangabe im neuen Namen in einen schon existierenden Ordner verlegt werden und umgekehrt.

PSAVE { PS } — Programm speichern (listgeschützt)

```
PSAVE "Programmname"
```

Für diesen Befehl gilt die gleiche Ausführung wie bei SAVE. Er bietet nur eine kleine Besonderheit, die aber in ihrer Wirkung sehr beeindruckend ist. Das P in diesem Befehlsnamen steht für protected (geschützt). Die GFA-Programme, die hiermit abgespeichert wurden:

- können nicht mehr gelistet werden.
- werden sofort nach dem Laden gestartet (Autostart, s. LOAD).

▸ können nicht bearbeitet werden. Wird versucht, mit dem Interpreter das - unsichtbare - Listing zu bearbeiten, wird man früher oder später mit einem Total-Absturz belohnt. Der Interpreter verfügt bei PSAVE-Programmen nicht mehr über die notwendigen Zeiger auf die Variablennamen, und beim Versuch, diese zu finden, gibt es Adressen-Wirrwarr.

RENAME { REN } — Datei umbenennen

```
RENAME "Name_alt" AS "Name_neu"
```

Entspricht exakt dem Befehl NAME (weiteres siehe dort).

RMDIR { RM } — Ordner löschen

```
RMDIR "Ordner"
```

Die Syntax dieses Befehls ist identisch mit der von CHDIR, nur daß der angegebene Ordner nicht geöffnet, sondern gelöscht wird. Befinden sich allerdings noch Dateien in dem Ordner, kann dieser Befehl nicht ausgeführt werden, und es erscheint eine Fehlermeldung. Ggf. sind die enthaltenen Dateien vorher durch KILL zu löschen.

SAVE { SA } — Programm speichern (codiert)

```
SAVE "Programmname"
```

SAVE speichert das im Programmspeicher befindliche Programm unter dem angegebenen Namen im Token-Format auf dem Festspeicher (Diskette/Hard-Disk) bzw. RAM-Disk (zur Pfad-

angabe siehe ggf. Kapitelanfang). Die Anführungsstriche zum Dateinamen können vernachlässigt werden, da sie vom Interpreter - falls nicht vorhanden - selbsttätig gesetzt werden. Die Extension zum Dateinamen kann ebenfalls vernachlässigt werden. Auch sie wird vom Interpreter gesetzt (.GFA - falls in Programmname keine andere Extension vorgegeben wurde). Befindet sich bereits ein Programm gleichen Namens auf der Diskette, wird es automatisch auf Programmname.BAK umbenannt.

5.5 Dateihandhabung

BGET { BG } — Teildatei lesen

```
BGET [#]Kanal,Start,Anz
```

Kanal ist der Identifikator einer mit OPEN "I" oder "U" geöffneten Datei. Ab File-Pointer-Position werden Anz Bytes dieser Datei in den Arbeitsspeicher - beginnend mit der Adresse Start - gelesen. Wird die Datei anschließend nicht mit CLOSE geschlossen, bleibt der File-Pointer auf der dem gelesenen Teil folgenden Byte-Position stehen. Der nächste Dateizugriff (BGET#, INP(#), INPUT#, PRINT#, OUT# etc.) bezieht sich dann auf diese Position. Beispiel:

```
Open "O",#1,"df0:Test.Dat"   ! Output-Datei öffnen
Print #1;"GFA-BASIC";        ! String hineinschreiben
Close                        ! und wieder schließen
A$=Space$(5)                 ! Kleinen Puffer setzen
Open "I",#1,"df0:Test.Dat"   ! Datei für Eingabe öffnen
Seek #1,4                    ! 4 Bytes überspringen
Bget #1,Varptr(A$),5         ! 5 Bytes in Puffer lesen
Close                        ! Datei schließen
Print A$                     ! String ausgeben
```

BPUT { BP } Teildatei schreiben

```
BPUT [#]Kanal,Start,Anz
```

Es werden Anz Bytes ab der Adresse Start gelesen und - beginnend mit der File-Pointer-Position - in eine mit OPEN "O" bzw. "U" geöffnete Datei mit dem Identifikator Kanal geschrieben. Im Gegensatz zu BSAVE können hiermit auch Teile einer Datei überschrieben oder an diese angefügt werden. Wird die Datei anschließend nicht mit CLOSE geschlossen, bleibt der File-Pointer auf der dem geschriebenen Teil folgenden Byte-Position stehen. Der nächste Dateizugriff (BGET#, INP(#), INPUT#, PRINT#, OUT# etc.) bezieht sich dann auf diese Position.

CLOSE { CL } Datenkanal schließen

```
CLOSE [#Kanal]
```

Alle mit OPEN geöffneten Kanäle müssen mit CLOSE #Kanal geschlossen werden, um ihre Kanal-Nummer für eine andere Datei verwenden zu können. Wird CLOSE ohne #Kanal verwendet, werden damit sämtliche Kanäle gleichzeitig geschlossen. Ein Kanal, der noch nicht mit CLOSE #Kanal geschlossen wurde und nochmals mit OPEN angesprochen wird, verursacht eine Fehlermeldung. Im Interpreter-Betrieb können nach Programmende von der Direkt-Ebene aus alle offenen Dateien weiterhin angesprochen werden, solange im Editor keine Programmänderung vorgenommen wurde.

EOF() — Datei auf Dateiende prüfen

```
Var=EOF(#Kanal)
```

Mit dieser Funktion kann festgestellt werden, ob sich der File-Pointer hinter dem letzten Byte der mit #Kanal angegebenen Datei - also am Dateiende (EndOfFile) - befindet. Ist dies der Fall, dann erhält man den Wert -1 (TRUE), andernfalls eine Null (FALSE). Die wohl wichtigste Anwendung dieser Funktion ist die Vermeidung der Fehlermeldung "File-Ende erreicht". Beispiel:

```
Open "I",#1,"Dateiname"  ! File zum Lesen öffnen
While Eof(#1)=False      ! Solange File-Ende nicht erreicht <--.
  Print Inp(#1)          ! Einzelnes Byte lesen und ausgeben   |
  Wend                   ! Wieder von vorn >-------------------'
```

FIELD { FI AS bzw. FI AT } — Datensatz in Felder unterteilen

```
FIELD #Kanal,Anz AS Var1$ [,Anz AS Var2$,...]
FIELD #Kanal,Anz AT(Adr1) [,Anz AT (Adr2),...]
```

Teilt die Datensätze der mit Kanal# angegebenen Random-Datei in so viele Einträge auf, wie durch die Menge der Anz AS Var$-Komponenten vorgegeben wird. Es wird jeweils die in Anz angegebene Anzahl an Bytes der nach AS folgenden String-Variablen (Var1$, Var2$ etc.) zugeordnet und die Variable gleichzeitig mit Leerzeichen gefüllt. Um numerische Daten nicht durch MKI$ etc. in Strings umwandeln zu müssen, kann die zweite Syntax-Variante verwendet werden. Dabei enthält Anz die Anzahl an Bytes, die ab der zugehörigen - hinter AT in Klammern gesetzten - Adresse gelesen werden sollen. Beispiel:

```
a%=673123
b%=VARPTR(a%)
c&=1000
d=61234.1231
FIELD #1,4 AT(b%),2 AT(*c&),8 AT(*d)
```

AT- und AS-Komponenten können in einer FIELD-Zeile beliebig gemischt werden (z.B. FIELD #1,20 AS a$,2 AT(*b&),...). Außerdem kann die FIELD-Aufteilung für eine Datei auf mehrere Programmzeilen verteilt sein. Diese werden dann als zusammengehörig angenommen. Siehe weiteres Beispiel unter 5.5.1 "Funktionsweise einer Random-Access-Datei".

GET # — Datensatz lesen

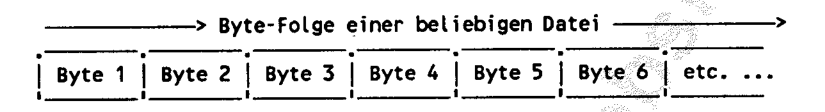
```
GET #Kanal [,Satznummer]
```

Kanal gibt die R-Datei (siehe OPEN) an, aus welcher der in Satznummer angegebene Datensatz gelesen werden soll. Bei der Zuordnung von Datensätzen zu einer R-Datei ist jedem Satz eine Nummer zuzuteilen. Durch deren Angabe kann der entsprechende Datensatz mit GET# wieder in die mit FIELD spezifizierten String-Variable(n) zurückgelesen werden. Fehlt "Satznummer", wird jeweils der nächste bzw. der durch RECORD ggf. gesetzte Datensatz gelesen. Beispiel unter 5.5.1 "Funktionsweise einer Random-Access-Datei".

LOC() — File-Pointer-Position

```
Var=LOC(#Kanal)
```

LOC (Abk. f. Location) liefert die aktuelle Position des Schreib- und Lesezeigers (File-Pointer) der durch #Kanal bestimmten Datei. Die gelieferte Byte-Position wird ab Dateianfang gezählt. Beispiel:




```
Angenommen: Aktueller Pointer zeigt auf Byte 4:
Pointer%=Loc(#1)   ! Liefert dann in Pointer% den Wert 4
```

LOF() — Dateilänge ermitteln

```
Var=LOF(#Kanal)
```

Durch LOF (Abk. f. Length Of File) ist es bei Disketten- bzw. Hard-Disk-Dateien möglich - sofern sie überhaupt einen Inhalt haben - ihre Byte-Länge zu ermitteln. Der Funktion wird dazu in Klammern die #Kanal-Nummer der betreffenden Datei übergeben. Beispiel:

```
Open "U",#1,"Dateiname"  ! Zuerst Datei öffnen
Print Lof(#1)            ! Variante 1: Direktausgabe
A%=Lof(#1)               ! Variante 2: Zuweisung
If Lof(#1)>32767         ! Variante 3: Bedingungsabfrage
  A$=Input$(32767,#1)
Else
  A$=Input$(Lof(#1),#1)  ! Variante 4: Einbindung
Endif
... etc.
```

OPEN { O } — Datenkanal öffnen

```
OPEN "Modus",#Kanal,"Dateiname" [,Satzlänge]
```

Dieser Befehl öffnet eine Datei - und Ihnen damit die bunte Welt der Dateiverwaltung. Seine Syntax ist auf den ersten Blick etwas kompliziert. Nach näherem Kennenlernen gewinnt man ihn jedoch schnell lieb. Die Möglichkeiten, die sich Ihnen mit den Befehlen zur Dateiverwaltung bieten, sind schier unüberschaubar.

Modus:

"O" Output öffnet eine Datei bzw. installiert sie neu. Existiert die angegebene Datei (auf Disk bzw. Hard-Disk) bereits, wird deren Länge auf Null gesetzt. In diesem Modus sind nur Schreibzugriffe zulässig.

"I" Input öffnet eine vorhandene Datei zum Herauslesen von Daten. Der File-Pointer wird auf das erste Byte der Datei gesetzt. Die Dateilänge bleibt durch Lesezugriffe unverändert. In diesem Modus sind nur Lesezugriffe zulässig.

"A" Append öffnet eine vorhandene Datei und setzt den File-Pointer auf das Dateiende. Alle an diese Datei auszugebenden Daten werden an das Dateiende angehängt (append = anhängen/hinzufügen).

"U" Update öffnet eine Datei für gleichzeitigen Schreib- und Lesezugriff. Dabei bleibt die ursprüngliche Länge erhalten. Es sei denn, bei der Datenausgabe wird über das aktuelle Dateiende hinaus geschrieben.

"R" Random öffnet eine Random-Access-Datei. Maximal sind 65535 Datensätze pro Datei möglich. Genauere Informationen hierzu finden Sie unter Kapitel 5.5.1. "Funktionsweise einer Random-Access-Datei".

#Kanal:

Es können gleichzeitig max. 100 Dateien geöffnet sein. Bei der Öffnung wird zur Identifikation jeder Datei eine Zahl zugewiesen. Unter Angabe dieser Zahl hinter dem Nummernzeichen kann nun z.B. durch Befehle wie PRINT#, INPUT#, BGET# etc. auf jede einzelne dieser Dateien zugegriffen werden. Die angegebene #Kanal-Nummer muß im Bereich von 0 bis 99 liegen.

Dateiname:

Bezeichnet den Namen der Datei, die mit dem betreffenden Zugriffsmodus geöffnet werden soll. Hier kann wieder die schon erwähnte Pfadstruktur (siehe Kapitelanfang) verwendet werden.

Ausnahme:

Sie können mit fast allen #-Dateibefehlen auch die verschiedenen Schnittstellen Ihres Amiga ansprechen. Dazu wird ebenfalls eine Quasi-Datei (virtuelle Datei) geöffnet. Mit Datei ist in diesem Fall jedoch nicht eine Disketten- bzw. Hard-Disk-Datei gemeint, sondern der jeweilige Port. Es stehen Ihnen vier solcher Ports zur Auswahl:

CON Console-Device

PRT Drucker-Port. Dadurch wird der in Preferences eingestellte Drucker angesprochen. Sämtliche Steuerzeichen werden durch den zuständigen Druckertreiber konvertiert.

PAR Parallele Schnittstelle. (Evtl. auch Druckeranschluß. Im Gegensatz zu PRT: keine Steuerzeichenkonvertierung.)

SER Serielle Schnittstelle mit den in Preferences eingestellten Parametern.

Der Ausdruck vor dem Gleichheitszeichen (inkl. Doppelpunkt) wird in diesem Fall als Dateiname an OPEN übergeben. Die Angabe von Modus kann bei Verwendung dieser Schnittstellen-Variante entfallen.

Satzlänge:

Mit Random-Access-Dateien ist es möglich, eine Reihe von Datensätzen innerhalb einer Datei anzulegen, die dann durch Identifikatoren angesprochen werden können. Der OPEN-Befehl erweitert sich ggf. hierfür um den Parameter Satzlänge. Darunter ist ein Wert zu verstehen, den Sie als Dateiverwalter selbst bestimmen können. Und zwar gibt er die Anzahl der Bytes an, die

Sie den Datensätzen einer Datei zukommen lassen wollen. Wird diese Angabe bei OPEN unterlassen, wird vom Interpreter eine Datensatzlänge von 128 Byte vorgesehen. Weitere Informationen zu diesem Thema finden Sie unter 5.5.1 "Funktionsweise einer Random-Access-Datei".

Die Eingabe der OPEN-Zeile kann extrem verkürzt werden. So wird z.B. o I 1 Name.Lst vom Editor in OPEN "I",#1,"Name.Lst" und z.B. o o 1 Name$ in OPEN "o",#1,Name$ umgewandelt. Alle An- und Ausführungsstriche, das Nummernzeichen sowie die Trennkommas können vernachlässigt werden, sofern zwischen den einzelnen Komponenten ein Leerzeichen angegeben wird.

PUT # { PU } — Datensatz schreiben

```
PUT #Kanal [,Satznummer]
```

Es wird der durch Satznummer definierte Datensatz aus den mit FIELD spezifizierten String-Variablen bzw. Adressen in die R-Datei mit der Nummer #Kanal geschrieben. Wird keine Satznummer angegeben, wird der jeweils nächste bzw. der durch RECORD ggf. gesetzte Datensatz gelesen. Genauere Informationen hierzu finden Sie unter 5.5.1 "Funktionsweise einer Random-Access-Datei".

RECALL { RECA } — String-Feld aus Datei lesen

```
RECALL #Kanal,Feld$(),.....,Zeilenvar
```

In Verbindung mit STORE# ein wahrlich gewaltiger Befehl, der es zum Vergnügen werden läßt, z.B. eine Textverarbeitung zu programmieren. Wer sich schon einmal mit einem solchen Programm angelegt hat, wird wissen, was ich meine.

RECALL liest Anz Textzeilen aus der zuvor geöffneten Datei mit der Kennung Kanal in das String-Feld Feld$(). Carriage Return (CHR$(13))/Line Feed (CHR$(10)) wird dabei als Zeilenendemarkierung interpretiert.

Den Elementen des Feldes werden der Reihe nach ab Element 0 (OPTION BASE 0) bzw. ab Element 1 (OPTION BASE 1) je eine Zeile zugeordnet. Ist das Feld zu kurz (die Elementeanzahl ist dann kleiner als die Anzahl gelesener Zeilen), wird der Leseprozeß beim letzten Element ohne Meldung abgebrochen. Wird das File-Ende erreicht, wird ebenfalls ohne Meldung abgebrochen. Zeilenvar ist eine numerische Rückgabevariable (Fließkomma- oder 32-Bit-Integer), die nach Abschluß die Anzahl der tatsächlich gelesenen Zeilen enthält.

Statt Anz kann auch der Ausdruck von TO Bis angegeben werden. Es werden dann die Zeilen ab dem Element von der Feldvariable bis zum Element Bis abgelegt.

Der Lesezeiger bleibt nach RECALL bei nicht geschlossenen Dateien auf dem Anfang der nächsten Zeile stehen. Das heißt, daß bei weiteren RECALL-Aufrufen ab dieser File-Position nur noch der verbleibende Dateirest gelesen wird. Soll wieder ab Datei-Beginn gelesen werden, ist der File-Pointer durch SEEK #Kanal,0 wieder auf den Datei-Anfang zu richten.

RECORD { REC } — Satz-Pointer für GET#/PUT# setzen

```
RECORD [#] Kanal,Satznummer
```

Setzt den Satz-Zeiger für den nächsten GET#- oder PUT#-Zugriff auf eine R-Datei. Wird beim nächsten GET# oder PUT# der Parameter "Satznummer" ausgelassen, wird der durch RECORD# aktualisierte Satz gelesen bzw. geschrieben.

RELSEEK { REL } File-Pointer verschieben

```
RELSEEK #Kanal,[-] Offset
```

RELSEEK (relativ Seek) verschiebt den File-Pointer der Datei mit der Kennung #Kanal relativ zur aktuellen Pointer-Position entweder in Richtung File-Ende oder File-Anfang um die mit "Offset" bestimmte Anzahl von Bytes. Im zweiten Fall (Richtung File-Anfang) ist dem Offset-Wert ein Minuszeichen voranzustellen. Beispiel:

```
Open "U",#1,"Dateiname"  ! Datei öffnen
Seek #1,Int(Lof(#1)/2)   ! Der File-Pointer zeigt nun
'                        ! auf das Byte in der Dateimitte
Relseek #1,3             ! Der Zeiger wird in Richtung File-Ende
'                        ! um drei Byte weitergesetzt
Relseek #1,-6            ! Der Zeiger wird in Richtung File-Anfang
'                        ! sechs Byte zurückgesetzt. Er befindet
'                        ! sich nun 3 Byte vor der durch SEEK
'                        ! bestimmten Mittelposition.
'... etc.                ! Weiteres Programm
```

Bei den Befehlen SEEK und RELSEEK ist darauf zu achten, daß nicht versucht wird, den Zeiger auf eine Position zu setzen, die in der angegebenen Datei nicht vorhanden ist, bzw. nicht vorhanden sein kann (kleiner Null oder größer File-Ende). Wird dies versucht, erscheint die Fehlermeldung "Seek falsch?".

SEEK { SEE } File-Pointer setzen

```
SEEK #Kanal,[-] Offset
```

Mit diesem Befehl kann der File-Pointer der Datei mit der Kennung "#Kanal" auf ein durch "Offset" bestimmtes beliebiges Byte gesetzt werden. Diese neue Position bezieht sich entweder absolut auf den File-Anfang oder auf das File-Ende. Im zweiten Fall

(relativ zum File-Ende) ist dem Offset-Wert ein Minuszeichen voranzustellen. Eine Anwendungsmöglichkeit finden Sie im RELSEEK-Beispiel.

TOUCH { TOU } Datei-Zeiteintrag aktualisieren

```
TOUCH #Kanal
```

Schreibt die aktuelle System-Zeitangabe (siehe TIME$) in die dafür vorgesehene Position des betreffenden Directory-Eintrags. #Kanal gibt dabei die Kennung der durch OPEN geöffneten Datei an.

STORE { STOR } String-Feld in Datei ablegen

```
STORE #Kanal,Feld$() [,Anz]
```

STORE speichert die Elemente eines durch Feld$() bestimmten String-Feldes der Reihe nach in der Datei mit der Kennung #Kanal. Als Endmarkierung wird jeder geschriebenen Zeile ein CR/LF (Carriage Return/Line Feed = CHR$(13)/CHR$(10)) angehängt. Durch den optionalen Parameter Anz kann bestimmt werden, wie viele Elemente maximal in der Datei gesichert werden sollen. Wird dieser Parameter ausgelassen, wird das komplette Feld gespeichert.

Statt Anz kann wie beim RECALL-Befehl auch der Ausdruck Von TO Bis angegeben werden, so daß nur ein Teilfeld abgespeichert wird.

5.5.1 Funktionsweise einer Random-Access-Datei

Wie die Behandlung von Variablenfeldern eine Sonderstellung einnimmt, so sind auch die Befehle zur Behandlung von Random-Access-Dateien (random access = wahlfreier Zugriff) eine besondere Erklärung wert.

Alle anderen Zugriffsvarianten auf Disketten- bzw. Hard-Disk-Dateien arbeiten grundsätzlich seriell, d.h. der Reihe nach. Es wird also eindimensional Zeichen für Zeichen nacheinander gelesen bzw. geschrieben. Diese Reihenfolge ist nur durch SEEK und RELSEEK veränderbar.

Eine R-Datei ist vergleichbar mit einem zweidimensionalen Speicherfeld. Es können mehrere Datenblöcke unter einem gemeinsamen Oberbegriff - hier der Dateiname - zusammengefaßt werden. Während die Elemente eines Variablenfeldes durch die Indizes repräsentiert werden, können hier die Datenblöcke (Elemente der 1. Dimension) durch die Satznummer angesprochen werden. Innerhalb eines Blocks befinden sich dann die zusammengehörigen Einträge (Elemente der 2. Dimension). Um eine solche Datei anzulegen, muß sie zuerst geöffnet werden. Ihr Sonderstatus wird dabei durch den Arbeitsmodus R kenntlich gemacht.

```
Open "R",#1,Dateiname,Satzlänge
```

Nachdem dieses getan wurde, kann mit dem Befehl FIELD bestimmt werden, in wieviel einzelne Einträge ein Satz unterteilt sein soll. Gleichzeitig wird angegeben, wie viele Zeichen (Bytes) jedem einzelnen dieser Einträge zuzuteilen sind. Da die Länge des gesamten Datensatzes durch die Angabe von Satzlänge im OPEN-Befehl bestimmt wird, ist es einleuchtend, daß die Summe der Zeichen der angegebenen Einträge insgesamt nicht größer sein darf als die Satzlänge. Sie darf aber kleiner sein. Die einmal so bestimmte Größe eines Satzes und seine Einteilung sollte von nun an nicht mehr verändert werden. Tun Sie es dennoch, ist die korrekte Übergabe der Daten an die zugeordneten Variablen nicht mehr gewährleistet. Natürlich können Sie neue Datenfelder in einen neu definierten Datensatz mit demselben

Dateinamen einsetzen, nur sind dann die vorher darin enthaltenen Daten nicht mehr zu erreichen.

Zum anderen dürfen die durch PUT# in die Feldeinteilung geschriebenen Datensätze in ihrer Größe nicht mehr verändert werden, da sonst Daten des folgenden Satzes mit Daten des aktuellen Satzes kollidieren. Um die festgelegten Satz- und Eintragslängen immer einzuhalten, werden zur Vorbereitung der Einträge die Befehle LSET und RSET verwendet. Diese fügen die übergebenen Texte in die definierten FIELD-String-Variablen ein, ohne deren Länge zu verändern.

Es ist problemlos möglich, eine installierte Random-Access-Datei zu erweitern oder zu ändern, indem man der Datei einfach weitere Sätze per PUT#-Anweisung anhängt bzw. in diese einfügt. Einmal geschriebene Sätze lassen sich jedoch nicht bzw. nur über Umwege wieder entfernen. Sie könnten z.B. ein solches File komplett über RECALL einlesen, dann durch INSERT oder DELETE bearbeiten und abschließend durch STORE wieder zurückschreiben.

Erweiterungen innerhalb des Satzgefüges (z.B. zusätzliche Einträge) sind nur möglich, indem eine neue Datei mit anderem FIELD-Aufbau eingerichtet wird und die Datensätze der alten Datei nach dem Eimerketten-Prinzip gelesen, erweitert und dann komplett an die neu eingerichtete R-Datei weitergeleitet werden.

Achten Sie darauf, daß bei jedem OPEN für eine schon eingerichtete Datei dieser Art immer exakt dieselbe Datensatzlänge angegeben wird bzw. die neu angegebene Satzlänge größer als die vorhandene ist. Wird nämlich keine Satzlänge genannt, wird diese vom Interpreter selbsttätig auf 128 Bytes festgelegt. Falls Sie die angesprochene Datei vorher größer dimensioniert hatten, wird beim Versuch, ein Feld anzusprechen, das mit seiner Länge über die angegebene Satzlänge hinausreicht, die Fehlermeldung "File-Ende erreicht" ausgegeben.

Eine Grafik soll das Prinzip verdeutlichen:

< 1. Datensatz > <'Satzlänge' z.B. 45 Byte>			< 2. Datensatz > < Satzlänge wie vor >			< n. Datensatz
Eintrag1 z.B. 10 Byte	Eintrag2 z.B. 20 Byte	Eintrag3 z.B. 15 Byte	Eintrag1 wie vor: 10 Byte	Eintrag2 wie vor: 20 Byte	Eintrag3 wie vor: 15 Byte	... —. ... |-> ... —'

Die Anzahl der Sätze, die Größe der einzelnen Einträge sowie die Satzlänge können von Ihnen frei bestimmt werden. Eingeschränkt werden Sie nur durch eine maximale Datensatzlänge von 65535 Byte und durch den Umstand, daß die Summe aller Eintragslängen die angegebene Satzlänge nicht überschreiten darf.

Zum tieferen Verständnis folgt nun ein Beispielprogramm, das Ihnen als Anregung zu eigenen Experimenten dienen soll. Dieses Programm besteht aus vier voneinander unabhängigen Blöcken. Im ersten Block wird eine Random-Access-Datei mit einer Satzlänge von 90 Byte installiert. Diese 90 Byte werden mit der FIELD-Anweisung in drei Einträge (40/30/20 Byte) unterteilt. Gleichzeitig werden die für die Aufnahme der Einträge vorgesehenen String-Variablen mit Leerzeichen gefüllt. Im Anschluß daran werden in der FOR/NEXT-Schleife sechs Datensätze eingelesen. Nach der Eingabe der jeweiligen Einträge werden die Inhalte der INPUT-Variablen durch LSET linksbündig in die vorgesehenen - und vorbereiteten - Puffer-Strings eingesetzt. Zu guter Letzt wird dann dieser Datensatz mit PUT# unter Angabe der Datensatznummer (hier der Schleifenindex I%) in die Datei geschrieben.

Block 2 demonstriert das Erweitern und Ändern einer bestehenden Datei. Hier sind die Texte willkürlich vorgegeben und können natürlich von Ihnen frei bestimmt werden. Die letzten beiden Programmblöcke zeigen zwei Möglichkeiten, die einzelnen Sätze aus der Datei wieder herauszulesen und auszugeben. Beispiel:

```
' Schreib-Variante 1:
' (Datei-Eröffnung)
'________________________
OPEN "R",#1,"Adress.Dat",90 ! R-Datei erstmals öffnen
FIELD #1,40 AS zeile.1$,30 AS zeile.2$,20 AS zeile.3$
'                           ! Satz einrichten
FOR i%=1 TO 6               ! 6 Sätze >──────────────.
  PRINT AT(20,11);"Name    :";                        |
  FORM INPUT 40,n$          ! Eintrag 1 eingeben      |
  PRINT AT(20,12);"Vorname :";                        |
  FORM INPUT 30,v$          ! Eintrag 2 eingeben      |
  PRINT AT(20,13);"Telefon :";                        |
  FORM INPUT 20,t$          ! Eintrag 3 eingeben      |
  CLS                       ! Bildschirm klar         |
  LSET zeile.1$=n$          ! Eintrag 1 linksbündig einfügen
  LSET zeile.2$=v$          !   -"-   2    -"-     -"-
  LSET zeile.3$=t$          !   -"-   3    -"-     -"-
  PUT #1,i%                 ! Datensatz in R-File eintragen
NEXT i%                     ! nächster Satz <────────'
CLOSE #1                    ! R-File schließen
'     Schreib-Variante 2 :
' (nachträgliches An-/Einfügen)
' ______________________________
OPEN "R",#1,"Adress.Dat",90 ! Öffnen (Satzlänge beachten)
FIELD #1,40 AS zeile.1$,30 AS zeile.2$,20 AS zeile.3$
'                           ! Satzeinteilung wie oben
LSET zeile.1$="Satz 7/Zeile 1" ! Eintrag 1 linksbündig
LSET zeile.2$="Satz 7/Zeile 2" !    -"-   2    -"-
LSET zeile.3$="Satz 7/Zeile 3" !    -"-   3    -"-
PUT #1,7                    ! neuen Datensatz nachtragen
LSET zeile.1$="Satz 2/Zeile 1" ! Eintrag 1 linksbündig
LSET zeile.2$="Satz 2/Zeile 2" !    -"-   2    -"-
LSET zeile.3$="Satz 2/Zeile 3" !    -"-   3    -"-
PUT #1,2                    ! 2. Datensatz überschreiben
CLOSE #1                    ! und wieder schließen
'   Lese-Variante 1:
' (seriell Einlesen)
' ______________________________
OPEN "R",#1,"Adress.Dat",90 ! Öffnen (Satzlänge beachten)
FIELD #1,40 AS zeile.1$,30 AS zeile.2$,20 AS zeile.3$
'                           ! Satzeinteilung wie oben
FOR i%=1 TO 7               ! 7 Datensätze >──────────.
  GET #1,i%                 ! lesen und die Einträge den
  '                         ! FIELD-Variablen zuweisen
  PRINT "Name      : ";zeile.1$ ! Eintrag 1 ausgeben  |
  PRINT "Vorname   : ";zeile.2$ !    -"-   2   -"-    |
  PRINT "Telefon   : ";zeile.3$ !    -"-   3   -"-    |
  PRINT                     ! Leerzeile               |
NEXT i%                     ! Nächster Satz <─────────'
CLOSE #1                    ! File schließen
CLS                         ! Bildschirm klar
PAUSE 200                   ! Kleine Pause
'   Lese-Variante 2:
```

```
' (wahlfrei Einlesen)
' ─────────────────────────────────
OPEN "R",#1,"Adress.Dat",90 ! Öffnen (Satzlänge beachten)
FIELD #1,40 AS eintrag1$,30 AS eintrag2$,20 AS eintrag3$
'                          ! Satzeinteilung wie gehabt
DO                         ! Lese-Schleife >─────────────.
  PRINT AT(20,11);"Satznummer eingeben (0=Abbruch): ";   |
  REPEAT                   ! Eingabe-Schleife >───────.  |
    INPUT satz%            ! Satznummer holen         |  |
  UNTIL satz%<8            ! Bis gültige Satznummer <--'  |
  EXIT IF satz%=0          ! A-Z-Taste oder 0 gedrückt?   |
  GET #1,satz%             ! Gewünschten Datensatz lesen  |
  PRINT AT(20,10);"Name    : ";eintrag1$ ! Eintrag 1 ausgeben|
  PRINT AT(20,11);"Vorname : ";eintrag2$ !   -"-   2   -"-  |
  PRINT AT(20,12);"Telefon : ";eintrag3$ !   -"-   3   -"-  |
  PRINT AT(20,15);"<Taste>"                                 |
  REPEAT                   ! Warte ... >─────.              |
  UNTIL LEN(INKEY$)        ! ... auf Taste <──'             |
  CLS                      ! Bildschirm klar                |
LOOP                       ! Wieder von vorn  <─────────────'
```

5.6 Port-Ein-/-Ausgabebefehle

INP — Daten byteweise von Peripherie lesen

```
Var=INP(#Kanal)
```

Diese Funktion dient dazu, ein einzelnes Byte aus einer Datei zu lesen. In der nachgestellten Klammer wird die Nummer des zugehörigen Datenkanals übergeben.

OUT { OU } — Daten byteweise an Peripherie ausgeben

```
OUT #Kanal,Byte1 [,Byte2 [,Byte3,...]]
```

Der Befehl OUT stellt das Gegenstück zu INP() dar. Hiermit lassen sich einzelne Byte-Werte an einen bestimmten Datenkanal senden. Beispiel:

```
Out #1,65
```

schreibt den ASCII-Wert des Zeichens A (ASCII 65) an die aktuelle File-Pointer-Position der Datei mit der Kanalnummer 1.

5.7 Die DOS-Bibliothek des Amiga

Die DOS-Bibliothek (englisch: DOS-Library) stellt - ebenso wie die anderen Bibliotheken des Amiga - eine Sammlung von Routinen dar, die dem Programmierer in jeder Programmiersprache zur Verfügung stehen. Die Routinen der DOS-Bibliothek sind hauptsächlich zur Vereinfachung des Zugriffs auf die verschiedenen Peripheriegeräte, speziell auf die Massenspeicher (Floppy, Harddisk usw.), implementiert worden.

Zum Glück enthält GFA-BASIC ja eine Vielzahl sehr komfortabler Ein-/Ausgabebefehle. Warum dann überhaupt noch mit der DOS-Bibliothek arbeiten? Nun, in einigen Situationen, zum Beispiel dann, wenn Sie das Directory einer Diskette analysieren möchten, kann die DOS-Bibliothek überaus nützlich sein, da GFA-BASIC keine vergleichbaren Befehle zur Verfügung stellt.

Auf der anderen Seite gibt es aber auch einige Routinen, die man in GFA-BASIC guten Gewissens vergessen kann, nicht zuletzt, weil sie zum Teil als wesentlich komfortablere GFA-BASIC-Befehle implementiert sind. Trotzdem möchte ich Ihnen diese Routinen nicht vorenthalten. Die Erklärungen beschränken sich in diesem Fall allerdings auf das Notwendigste. Nähere Informationen zu diesen Routinen (und natürlich auch zu allen anderen) finden Sie zum Beispiel in Amiga Intern, Intern 2 oder im großen AmigaDOS-Buch, beide von DATA BECKER.

Wie kann man nun auf die Routinen der DOS-Bibliothek zugreifen? Dies gestaltet sich - im Gegensatz etwa zu AmigaBASIC - erfreulich einfach: Alle Routinen stehen (ohne irgendwelche Öffnungszeremonien) jederzeit als GFA-BASIC-Funktionen zur Verfügung!

Vom GFA-BASIC-Editor wird sogar die korrekte Syntax der Funktionen überprüft. Fehlermeldungen des Betriebssystems (aufgrund semantisch unkorrekter Angaben) können in der Regel allerdings nicht abgefangen werden, d.h. Sie bekommen entweder eine AmigaDOS-Fehlermeldung oder - im schlimmsten Fall - eine sog. Guru-Meditation, nach der in der Regel nur noch ein Reset hilft. Also Vorsicht! Doch schauen wir uns die Routinen im einzelnen an; 31 sind es insgesamt.

Anmerkung: Bei allen Funktionen, die keine Rückgabewerte liefern, verwenden Sie zum Aufruf am besten den Befehl VOID (siehe auch Kapitel 12).

Allgemeine Ein-/Ausgabefunktionen

Open ________________ Datei öffnen

```
Filehandle = Open (Name, Modus)
```

Öffnet die Datei, auf deren Name der Parameter 'Name' zeigt. Achtung: Der Namensstring muß mit einem Nullbyte abgeschlossen sein! Diese sog. C-Konvention für Strings gilt für praktisch alle Stringübergaben an Betriebssystemfunktionen.

Modus enthält wahlweise den Wert 1005 (eine bestehende Datei wird zum Schreiben oder Lesen geöffnet), 1006 (eine neue Datei wird erzeugt und zum Schreiben geöffnet. Achtung: Eine evtl. bereits existierende Datei gleichen Namens wird dabei überschrieben!) oder 1004 (wie 1005, aber diese Datei kann dann nicht mehr von anderen Programmen geöffnet werden, solange bis diese wieder geschlossen wird; Sie haben dann also exklusiven Zugriff auf diese Datei).

In Filehandle wird ein Zeiger auf die sog. Filehandle-Struktur der Datei übergeben. Diese enthält wichtige Informationen über die Datei, die aber im Normalfall nur für die interne Verwaltung von AmigaDOS von Bedeutung sind. Konnte die Funktion nicht ausgeführt werden, enthält Filehandle den Wert Null.

Bei den weiteren Dateifunktionen hat Filehandle dieselbe Funktion wie die Dateinummer in GFA-BASIC: Sie dient zur Kennzeichnung und leichteren Identifizierung der Datei.

Close — Datei schließen

```
VOID Close (Filehandle)
```

Schließt die mit Open geöffnete Datei. Filehandle enthält den bei Open erhaltenen Zeiger auf die Filehandle-Struktur.

Read — Daten lesen

```
Anzahl = Read (Filehandle, Puffer, Länge)
```

Read liest aus der durch Filehandle bezeichneten Datei bis zu 'Länge' Bytes und legt sie im Speicher ab Adresse 'Puffer' ab. In 'Anzahl' wird die Anzahl der tatsächlich gelesenen Bytes zurückgegeben. Ist dieser Wert gleich -1, so trat während des Lesevorgangs ein Fehler auf.

Wrte — Daten schreiben

```
Anzahl = Write (Filehandle, Puffer, Länge)
```

Write schreibt in die durch Filehandle bezeichnete Datei 'Länge' Bytes aus dem Speicher ab der Adresse 'Puffer'. In 'Anzahl' wird die Anzahl der tatsächlich geschriebenen Bytes zurückgegeben. Ist dieser Wert gleich -1, so trat während des Schreibvorgangs ein Fehler auf.

Seek — Datenzeiger positionieren

```
Position = Seek (Filehandle, Abstand, Modus)
```

Seek positioniert bzw. verstellt den internen Zeiger der mit Filehandle bezeichneten Datei. Dieser Zeiger zeigt jeweils auf das nächste zu lesende oder zu schreibende Byte der Datei.

'Modus' gibt an, ob der in 'Abstand' angegebene Wert den Zeiger relativ zum Dateianfang (Modus=-1), zur aktuellen Position (Modus=0) oder zum Dateiende (Modus=1) verstellen soll. Dabei sind in 'Abstand' auch negative Werte zugelassen; der Zeiger wird dann rückwärts verschoben.

In 'Position' wird die nach Ausführung der Funktion aktuelle Position des Zeigers zurückgegeben. Um die momentane Position des Zeigers zu erhalten, stellt man 'Modus' einfach auf 0 (=Verschiebung relativ zur aktuellen Position) und gibt als Abstand 0 Bytes an:

Input — Standard-Eingabekanal ermitteln

```
Position=Seek(Filehandle,0,0)
```

Filehandle = Input()

Gibt das Filehandle des Standard-Eingabekanals zurück.

Output — Standard-Ausgabekanal ermitteln

```
Filehandle = Output()
```

Gibt das Filehandle des Standard-Ausgabekanals zurück.

WaitForChar — Auf Empfang eines Zeichens warten

```
Status = WaitForChar (Filehandle, Timeout)
```

WaitForChar wartet die in Timeout angegebene Anzahl von Mikrosekunden auf den Empfang eines Zeichens aus der mit Filehandle bezeichneten Datei.

Wird in dieser Zeit kein Zeichen empfangen, so erhält Status den Wert 0, andernfalls den Wert -1. Das Zeichen kann dann mit der Funktion Read ausgelesen werden.

IsInteractive ______________________ Kanal-Typ ermitteln

```
Status = IsInteractive (Filehandle)
```

IsInteractive ermittelt, ob es sich bei der durch Filehandle bezeichneten Datei um eine interaktive Datei (Ein- und Ausgaben sind möglich) handelt (Status=-1) oder nicht (Status=0).

IoErr ______________________ Ein-/Ausgabefehlernummer ermitteln

```
Fehler = IoErr ()
```

In 'Fehler' wird die Nummer des zuletzt (bei der Abarbeitung einer der anderen Funktionen) aufgetretenen Fehlers zurückgegeben.

Bei den anderen Funktionen selbst wird ein aufgetretener Fehler meist durch eine Null als Rückgabewert signalisiert. Mit Hilfe von IoErr erfährt man dann die genaue Fehlerursache.

Disketten- und Dateiorganisation

Lock ______________________ Zugriffsstatus bestimmen

```
Lockstruktur = Lock (Name, Modus)
```

Lock sucht nach einer Datei oder einem Unterdirectory mit dem Namen, auf den 'Name' zeigt, und erzeugt dafür eine sog. Lock-Struktur, deren Adresse in der Variablen 'Lockstruktur' übergeben wird. Diese Lock-Struktur (bzw. deren Adresse) wird bei den folgenden Funktionen benötigt, um auf eine Datei (oder ein Unterdirectory) überhaupt zugreifen zu können.

'Modus' gibt an, welcher Art der Zugriff auf die betr. Datei sein soll. Modus=-2 bedeutet, daß aus der Datei von mehreren Programmen aus gelesen werden kann; Modus=-1 bestimmt, daß die Datei nur vom aktuellen Programm aus beschrieben werden kann.

DupLock — Lock-Struktur kopieren

```
Lockstruktur2 = DupLock (Lockstruktur1)
```

DupLock kopiert eine durch ihre Adresse bezeichnete Lock-Struktur ('Lockstruktur1') und übergibt in 'Lockstruktur2' die Adresse der Kopie. Wichtig: Es können nur Lock-Strukturen kopiert werden, die zum Lesen zugelassen sind (Modus=-2 bei Lock, siehe dort).

UnLock — Lock-Struktur entfernen

```
VOID UnLock (Lockstruktur)
```

UnLock entfernt eine mit Lock oder DupLock erzeugte Lock-Struktur aus dem Speicher. In 'Lockstruktur' wird dazu die Adresse der Lock-Struktur übergeben.

Haben Sie bis hierher durchgehalten? Ja, dann werden Sie gleich sehen, daß es sich gelohnt hat. Während die zuvor besprochenen Funktionen doch wohl mehr etwas für den Spezialisten waren, bringen Ihnen die nun folgenden Funktionen ganz handfeste Informationen, die man so direkt von GFA-BASIC aus nicht erhalten kann.

Info — Disketteninformationen holen

```
Status = Info (Lockstruktur, Infoadresse)
```

Info erzeugt im Speicher ab 'Infoadresse' einen Infoblock mit diversen Informationen über die durch 'Lockstruktur' bezeichnete Diskette. 'Lockstruktur' muß entweder auf den Namen der Diskette oder auf eine Datei oder ein Unterdirectory der Diskette zeigen. Anstelle des Diskettennamens kann auch eine Laufwerksbezeichnung genommen werden (siehe Beispiel).

Achtung: Bei 'Infoadresse' muß es sich unbedingt um eine Adresse handeln, die durch 4 teilbar ist! Zur Reservierung des Speicherplatzes für die Infotabelle sollten Sie daher immer die Funktion MALLOC verwenden, da durch MALLOC reservierte Bereiche grundsätz-

lich an einer durch 8 teilbaren Adresse beginnen. Der durch Info erzeugte Infoblock hat folgenden Aufbau:

Byte	Bedeutung
0-3	Anzahl der Diskettenfehler
4-7	installierte Disketteneinheit
8-11	Diskettenstatus: 80 Diskette ist schreibgeschützt / 82 Diskette ist beschreibbar.
12-15	Gesamtanzahl der Diskblöcke
16-19	Anzahl der belegten Blöcke
20-23	Anzahl der Bytes je Block
24-27	Diskettentyp: -1 Keine Diskette eingelegt / BAD Diskette unlesbar / DOS DOS-Diskette
28-31	Zeiger auf Diskettenna@en
32	Diskette aktiv (< >0) / inaktiv (0)

Um an die einzelnen Informationen zu gelangen, verwenden Sie entweder PEEK und Co. oder einen der Speicherzugriffsbefehle aus Kapitel 12 (CARD{} usw.). Am interessantesten dürfte die Möglichkeit sein, zu erfahren, ob eine in einem Laufwerk eingelegte Diskette schreibgeschützt ist. Zwar reagiert AmigaDOS mit einem entsprechenden System-Requester auf einen Schreibzugriff auf eine Diskette mit aktiviertem Schreibschutz. Wenn man den Schutzstatus aber schon vor dem Schreibzugriff ermittelt, kann man natürlich ganz individuell darauf reagieren. Zu diesem Zweck habe ich die folgende Funktion geschrieben:

```
FUNCTION disktest(drive$)          !drive$ enthält Laufwerksbezeichnung
  LOCAL drive%,lock%,adr%,status% !lokale Variablen
  drive$=drive$+CHR$(0)            !String muß mit 0 enden
  drive%=VARPTR(drive$)            !Zeiger auf Stringanfang
  lock%=Lock(drive%,-2)            !Lockstruktur (lock% enthält Adresse)
  adr%=MALLOC(36,1)                !36 Byte für Infostruktur reservieren
  VOID Info(lock%,adr%)            !Info-Funktion aufrufen
  status%=PEEK(adr%+11)            !Diskettenstatus holen
  VOID MFREE(adr%,36)              !reservierten Speicher freigeben
  IF status%=82 THEN               !Diskette beschreibbar?
    RETURN TRUE                    !ja, dann -1 zurückgeben
  ELSE                             !Diskette schreibgeschützt
    RETURN FALSE                   !0 zurückgeben
  ENDIF
ENDFUNC
```

Wie nutzt man nun 'Disktest'? Nehmen wir einmal an, Sie möchten feststellen, ob die in Laufwerk DF0: eingelegte Diskette beschreibbar ist. Das könnte dann zum Beispiel so aussehen:

```
IF @disktest("DF0:")=FALSE THEN
  ALERT 0,"Bitte Schreibschutz entfernen!",1,"OK",d%
ENDIF
' Schreibzugriff .....
```

CreateDir neues Verzeichnis erzeugen

```
Lockstruktur = CreateDir (Name)
```

Erstellt im aktuellen Directory ein neues Unterdirectory mit dem Namen, auf den 'Name' zeigt. Dabei wird - analog zu der Funktion Lock - eine Lock-Struktur erzeugt, deren Adresse in 'Lockstruktur' zurückgegeben wird.

CurrentDir __________ Aktuelles Verzeichnis bestimmen

```
alte_Lockstruktur = CurrentDir (Lockstruktur)
```

Erhebt das durch seine Lock-Struktur bezeichnete Unterdirectory zum aktuellen Directory. Zurückgegeben wird die Adresse der Lock-Struktur des zuvor aktuellen Directories.

ParentDir __________ Übergeordnetes Verzeichnis ermitteln

```
neue_Lockstruktur = ParentDir (Lockstruktur)
```

Ermittelt das übergeordnete Directory des durch seine Lock-Struktur bezeichneten Unterdirectories und übergibt die Adresse dessen Lock-Struktur.

Examine __________ Fileinformationen holen

```
Status = Examine (Lockstruktur, Infoadresse)
```

Examine erzeugt im Speicher ab 'Infoadresse' einen Infoblock mit diversen Informationen über die/das durch 'Lockstruktur' bezeichnete Datei/Directory.

Der Infoblock hat folgenden Aufbau:

Byte	Bedeutung
0-3	Diskettennummer
4-7	Eintragstyp (>0: Directory; sonst: Datei)
8-115	Eintragsname (nur 30 Byte nutzbar)
116-119	Zugriffsstatus (siehe auch Funktion SetProtection)
Bit 0:	Datei nicht löschbar
Bit 1:	Datei nicht ausführbar
Bit 2:	Datei nicht überschreibbar
Bit 3:	Datei nicht lesbar
120-123	Eintragstyp (siehe oben)
124-127	Dateilänge in Bytes
128-131	Anzahl der belegten Blocks
132-143	Erstellungsdatum
144-259	Kommentar (nur 80 Byte nutzbar)

Um die Informationen auszulesen, gehen Sie wie folgt vor:

```
' dname$ enthält Directory-/Dateinamen
dname$=dname$+CHR$(0)          !String muß mit 0 enden
dname%=VARPTR(dname$)          !Zeiger auf Stringanfang
lock%=Lock(dname%,-2)          !Lockstruktur erzeugen
infoadr%=MALLOC(260,1)         !260 Byte für Infoblock reservieren
VOID Examine(lock%,infoadr%)   !Funktion aufrufen
```

Nun können Sie leicht auf die einzelnen Daten zugreifen:

```
laenge%={infoadr%+124}  !Dateilänge
kom$=CHAR{infoadr%+144} !Kommentar   usw.
```

Zum Schluß vergessen Sie bitte nicht, den durch MALLOC belegten Speicher mit

```
VOID MFREE(infoadr%,260)
```

wieder freigeben zu lassen! Examine läßt sich übrigens auch dazu verwenden, den Namen einer Diskette zu ermitteln. Das kann zum Beispiel dann sehr nützlich sein, wenn Sie festellen wollen, ob die richtige Diskette in das Laufwerk eingelegt ist. Dazu übergeben Sie in dname$ einfach die Laufwerksbezeichnung (z.B. DF0:). Mit

```
disk$=CHAR{infoadr%+8}
```

erhalten Sie dann den Namen der Diskette.

ExNext ______ Nächsten Directory-Eintrag ermitteln

```
Status = ExNext (Lockstruktur, Infoadresse)
```

ExNext ermittelt den jeweils nächsten Eintrag eines durch seine Lock-Struktur bezeichneten Directories und legt die Informationen über diesen Eintrag in dem durch 'Infoadresse' bezeichneten Infoblock ab (siehe Examine). Mit ExNext läßt sich nach und nach das gesamte Inhaltsverzeichnis einer Diskette auslesen. Ist kein weiterer Eintrag vorhanden, wird in 'Status' eine Null zurückgegeben.

DeleteFile ______ Datei löschen

```
Status = DeleteFile (Name)
```

Löscht eine Datei oder ein Unterdirectory. ('Name' enthält die Adresse des Namens.) Wichtig: Ein zu löschendes Unterdirectory darf keine Dateien mehr enthalten! Diese müssen gegebenenfalls zuvor - mit DeleteFile - einzeln gelöscht werden.

Rename ______ Datei umbenennen

```
Status = Rename (alter_Name, neuer_Name)
```

Benennt die Datei oder das Directory 'alter_Name' in 'neuer_Name' um.

SetProtection ______ Dateistatus setzen

```
Status = SetProtection (Name, Modus)
```

Verändert den Zugriffsstatus der Datei oder des Unterdirectories 'Name'. In 'Modus' haben nur die untersten vier Bits eine Bedeutung (jeweils, wenn gesetzt):

Bit 0: Datei nicht löschbar
Bit 1: Datei nicht ausführbar
Bit 2: Datei nicht überschreibbar
Bit 3: Datei nicht lesbar

Beispiel:

```
dname$="Dateiname"+CHR$(0)          !beliebiger Name
dname=VARPTR(dname$)                !Zeiger darauf
modus=2^0 OR 2^2                    !Datei nicht löschbar/nicht
                                     überschreibbar
Status=SetProtection(dname,modus)  !Modus setzen
```

Trat kein Fehler auf, enthält 'Status' anschließend den Wert TRUE (-1).

SetComment — Dateikommentar setzen

```
Status = SetComment (Name, Kommentar)
```

Versieht die Datei oder das Unterdirectory 'Name' mit einem Kommentar, der bis zu 80 Zeichen lang sein darf. Beispiel:

```
dname$="Dateiname"+CHR$(0)     !beliebiger Name
dname=VARPTR(dname$)           !Zeiger darauf
kom$="Kommentar"+CHR$(0)       !beliebiger Kommentar bis 80 Zeichen
kom=VARPTR(kom$)               !Zeiger darauf
status=SetComment(dname,kom)   !Kommentar setzen
```

Lief alles glatt, so enthält 'Status' anschließend den Wert TRUE (-1). Zum Lesen der an eine Datei angehängten Kommentare verwenden Sie das CLI-Kommando List. List muß dazu mit dem GFA-BASIC-Befehl EXEC aufgerufen werden. Zum Beispiel listet EXEC "list df0:",0,0 alle Dateien der Diskette im Laufwerk df0: samt ihren evtl. vorhandenen Kommentaren auf. Die Ausgabe erfolgt dabei in ein CLI-Fenster (nicht in das GFA-BASIC-Ausgabefenster).

Prozeß-Organisation

CreateProc — Neuen Prozeß erstellen

```
Prozess = CreateProc(Name, Prior, Segment, Stack)
```

Erstellt unter dem Namen, auf den 'Name' zeigt, eine neue Prozeß-Struktur.

DeviceProc — Den I/O verwendenden Prozeß ermitteln

```
Prozeß = DeviceProc(Name)
```

Übergibt in 'Prozeß' die Identifikationsnummer des Prozesses, der im Moment den mit 'Name' angegebenen Ein-/Ausgabekanal verwendet.

Delay — aktuellen Prozeß anhalten

```
VOID Delay (Zeit)
```

Hält den aktuellen Prozeß um die in 'Zeit' angegebene Anzahl von 1/50 Sekunden an.

Exit — Prozeß beenden

```
VOID Exit (Parameter)
```

Beendet den laufenden Prozeß (bzw. das laufende Programm) und gibt die dem Prozeß zugeteilten Speicherbereiche wieder frei.

LoadSeg — Programmdatei laden

```
Segment = LoadSeg (Name)
```

Lädt die Programmdatei 'Name' in den Speicher. In 'Segment' wird die Adresse des ersten Programm-Moduls zurückgegeben.

UnLoadSeg ________ **Geladene Programmdatei löschen**

```
VOID UnLoadSeg (Segment)
```

Löscht eine mit LoadSeg geladene Programmdatei und gibt den verwendeten Speicherplatz wieder frei. 'Segment' enthält die Adresse des ersten Programm-Moduls (siehe auch LoadSeg).

DateStamp ________ **Datum und Uhrzeit ermitteln**

```
VOID DateStamp (Adresse)
```

Übergibt in drei Langworten (auf die 'Adresse' zeigt) das aktuelle Datum im AmigaDOS-Format.

Execute ________ **CLI-Kommando ausführen**

```
Status = Execute (Kommando, Input, Output)
```

Execute führt ein beliebiges CLI-Kommando aus. Dazu wird in 'Kommando' ein Zeiger auf den Kommandotext übergeben. Input und Output enthalten das Filehandle des gewünschten Ein- bzw. Ausgabekanals.

Anmerkung: Die DOS-Funktionen GetPacket und QueuePacket sind im Augenblick von GFA-BASIC aus nicht direkt ansprechbar.

5.8 Drucker-Anweisungen

HARDCOPY { H } (Teil-) **Bildschirm auf Drucker ausgeben**

```
HARDCOPY
HARDCOPY Rastport, Colormap, Modi, Abst_l, Abst_r, Breite, Höhe, Spalten, Zeilen, Flags
HARDCOPY Flags
```

HARDCOPY erzeugt einen Grafikausdruck des aktuellen Screens auf dem Drucker. Dabei werden die Druckereinstellungen aus

den Preferences verwendet. Damit HARDCOPY einwandfrei arbeitet, muß auf dem betr. Screen mindestens ein Fenster geöffnet sein. Die zweite und dritte Syntaxversion erlaubt es, den Ausdruck auf vielfältigste Art und Weise zu beeinflussen. Schauen wir uns die Parameter im einzelnen an:

Rastport: Adresse des Rastports, von dem die Hardcopy gemacht werden soll.

Colormap: Adresse der Colormap, in der die Farben des Bildes enthalten sind.

Modi: Enthält die Auflösungsmodi, wie in der Viewport-Struktur angegeben.

Abst_l: Abstand zum linken Rand des Rastports.

Abst_r: Abstand zum rechten Rand des Rastports.

Breite: Breite des auszudruckenden Bereichs.

Höhe: Höhe des auszudruckenden Bereichs.

Spalten: Anzahl der zu druckenden Spalten.

Zeilen: Anzahl der zu druckenden Zeilen.

Flags: Hier sind nur die untersten 12 Bit des Wertes von Interesse. Jedes Bit hat eine andere Bedeutung:

Bit	Bedeutung, wenn gesetzt
0	Der Wert in 'Spalten' gilt als 1/1000- Zoll- Angabe.
1	Der Wert in 'Zeilen' gilt als 1/1000- Zoll-Angabe.
2	Es wird in Maximalbreite gedruckt.
3	Es wird in Maximalhöhe gedruckt.
4	Der Wert in 'Spalten' wird als ein Teiler der Maximalbreite angesehen.
5	Der Wert in 'Zeilen' wird als ein Teiler der Maximalhöhe angesehen.
6	Bewirkt eine zentrierte Ausgabe des Druckbildes.
7	Das Seitenverhältnis auf dem Papier soll dem Seitenverhältnis auf dem Bildschirm entsprechen.
8-10	Mit diesen Bits läßt sich festlegen, in welcher Auflösung die Hardcopy ausgegeben werden soll. '001' druckt in der niedrigsten Auflösung des Druckers, '111' in der höchsten.
11	Der Seitenvorschub am Ende der Ausgabe wird unterdrückt. Dadurch lassen sich mehrere Hardcopies 'nahtlos' aneinanderfügen.

Anmerkung: Was ein Rastport, ein Viewport oder eine Colormap ist und wie Sie deren Adressen und sonstigen Werte ermitteln können, erfahren Sie im Grafikkapitel. Beispiel:

```
OPENW 5                 !Kleines, zentriertes BASIC-Fenster
GRAPHMODE 1             !Grafikmodus einstellen
ELLIPSE 150,160,50,40   !Zwei Ellipsen
ELLIPSE 150,160,80,25   !zeichnen
HARDCOPY                !Workbench-Screen ausdrucken
CLOSEW 5                !Fenster schließen
```

Das Programm zeichnet eine kleine Grafik auf den Bildschirm und bringt dann den gesamten Workbench-Screen zu Papier.

LLIST { LL } — Programm-Listing ausdrucken

```
LLIST
```

Soll das aktuelle Programm-Listing auf den Drucker übertragen werden, kann das durch LLIST erreicht werden (vgl. Editor-Funktion Llist). Die Ausführung dieses Befehls kann durch die GFA-Break-Funktion (<Control/Shift/Alternate>) unterbrochen werden. Der Interpreter ist dann wieder arbeitsbereit, und es wird nur noch der Inhalt des Druckerpuffers ausgedruckt.

Die im Programm befindlichen Punkt-Befehle (siehe Erklärung zur Editor-Funktion Llist) werden auch hier berücksichtigt.

LPOS() — Druckkopfposition ermitteln

```
Var=LPOS(Dummy)
```

Mit dieser Funktion können Sie den aktuellen Standort des virtuellen Druckkopfes im Zeichenpuffer Ihres Druckers ermitteln (max. 255). Da dieser nicht mit der aktuellen Position des

physikalischen Schreibkopfes (der, der den Lärm macht) übereinstimmen muß, kann es unter Umständen wichtig sein, die gerade im Speicher aktuelle Position zu erfahren. Nach einem Carriage Return (LPRINT) beginnt die Zählung wieder bei Null. Es wird in Klammern ein beliebiger Wert übergeben, der für die Funktion ohne Bedeutung ist (dummy; engl. für: Attrappe/Schwindel). Beispiel:

```
Lprint "VIRTUELLE DRUCKKOPF-POSITION"
For I=1 To 150
  Lprint Lpos(0)'
Next I
A=Lpos(0)
Print A
```

LPRINT { LPR } — Daten an Drucker ausgeben

```
LPRINT [,'] "Text" [[;,'] Var1 [;,'] Expr...]
```

Grundsätzlich arbeitet LPRINT genauso wie PRINT. Die Syntax ist bis auf die AT(Xpos,Ypos)-Variante mit PRINT vergleichbar. Ein Unterschied ist, daß LPRINT die gewünschten Zeichen, Strings und Werte nicht auf dem Bildschirm, sondern auf dem Drucker ausgibt. Zu einem leidigen Problem bei der Druckeransteuerung wird häufig die Einstellung der verschiedenen Sonderfunktionen mit Hilfe sogenannter Steuerzeichen. Zwar halten sich mittlerweile die meisten Drucker an den sog. EPSON ESC/P-Standard, trotzdem kommt es aber immer wieder vor, daß der Drucker nicht das macht, was er eigentlich tun sollte. Etwa wenn es darum geht, auf einen bestimmten Zeichensatz umzuschalten oder einen anderen Zeilenabstand einzustellen.

Im Vergleich zu anderen Rechnern hat man es hier beim Amiga etwas leichter: Der Amiga verfügt über eine Standard-Steuerzeichen-Tabelle. Diese Standard-Steuerzeichen werden - sofern sie über das logische Gerät PRT: gesendet werden, was bei LPRINT der Fall ist - von der in den Preferences eingestellten Druckeranpassung in die Steuercodes des jeweils angewählten

Druckers übersetzt. Der Vorteil liegt klar auf der Hand: Solange Sie sich beim Programmieren nur an die Standard-Steuercodes halten, ist es völlig egal, welcher Drucker jeweils an dem Amiga angeschlossen ist. Hauptsache, die richtige Druckeranpassung ist eingestellt.

Eines sollten Sie aber immer beachten: Der angeschlossene Drucker muß auch in der Lage sein, die mit dem betreffenden. Steuerzeichen verbundene Aktion auszuführen. Bei einem Drucker, der zum Beispiel über keinen französischen Zeichensatz verfügt, nützt natürlich auch das entsprechende Standard-Steuerzeichen nur sehr wenig. Bitte informieren Sie sich also gegebenenfalls in Ihrem Druckerhandbuch, über welche Sonderfunktionen Ihr Drucker verfügt.

Welche Standard-Druckersteuerzeichen gibt es nun? Schauen Sie dazu bitte in die folgende Tabelle. Sie enthält eine Übersicht aller verfügbaren Steuersequenzen:

CHR$(27)+"c"	Drucker initialisieren
CHR$(27)+"#1"	Abschaltung aller Sondermodi
CHR$(27)+"D"	Zeilenvorschub
CHR$(27)+"E"	Zeilenvorschub+Wagenrücklauf
CHR$(27)+"M"	eine Zeile zurück
CHR$(27)+"[0m"	normale Zeichendarstellung
CHR$(27)+"[1m"	Fettdruck ein
CHR$(27)+"[22m"	Fettdruck aus
CHR$(27)+"[3m"	Kursiv ein
CHR$(27)+"[23m"	Kursiv aus
CHR$(27)+"[4m"	Unterstreichen ein
CHR$(27)+"[24m"	Unterstreichen aus
CHR$(27)+"["+x$+"m"	Vordergrundfarbe (x$ zwischen " 30" und "39"); Hintergrundfarbe (x$ zwischen "40" und "49")
CHR$(27)+"[0w"	normale Schriftgröße
CHR$(27)+"[2w"	Elite-Schrift ein
CHR$(27)+"[1w"	Elite-Schrift aus
CHR$(27)+"[4w"	Schmalschrift ein
CHR$(27)+"[3w"	Schmalschrift aus
CHR$(27)+"[6w"	Breitschrift ein
CHR$(27)+"[5w"	Breitschrift aus
CHR$(27)+"[2"+CHR$(34)+"z"	NLQ ein
CHR$(27)+"[1"+CHR$(34)+"z"	NLQ aus

CHR$(27)+"[4"+CHR$(34)+"z"	Doppeldruck ein
CHR$(27)+"[3"+CHR$(34)+"z"	Doppeldruck aus
CHR$(27)+"[6"+CHR$(34)+"z"	Schattenschrift ein
CHR$(27)+"[5"+CHR$(34)+"z"	Schattenschrift aus
CHR$(27)+"[2v"	Superscript ein
CHR$(27)+"[1v"	Superscript aus
CHR$(27)+"[4v"	Subscript ein
CHR$(27)+"[3v"	Subscript aus
CHR$(27)+"L"	Hochstellen (Halbschritt)
CHR$(27)+"K"	Tiefstellen (Halbschritt)
CHR$(27)+"[0v"	zurück zu Normalschrift
CHR$(27)+"[2p"	Proportionalschrift ein
CHR$(27)+"[1p"	Proportionalschrift aus
CHR$(27)+"[0p"	Proportionalabstand löschen
CHR$(27)+"["+x$+"E"	Proportionalabstand = x$
CHR$(27)+"[5F"	links ausrichten
CHR$(27)+"[7F"	rechts ausrichten
CHR$(27)+"[6F"	Blocksatz
CHR$(27)+"[0F"	Blocksatz aus
CHR$(27)+"[3F"	Buchstabenbreite justieren
CHR$(27)+"[1F"	zentrieren
CHR$(27)+"[0z"	Zeilenabstand 1/8 Zoll
CHR$(27)+"[1z"	Zeilenabstand 1/6 Zoll
CHR$(27)+"["+x$+"t"	Seitenlänge auf x$ Zeilen einstellen
CHR$(27)+"["+x$+"q"	Perforation um x$ Zeilen überspringen
CHR$(27)+"[0q"	Perforation überspringen aus
CHR$(27)+"(B"	amerikanischer Zeichensatz
CHR$(27)+"(R"	französischer Zeichensatz
CHR$(27)+"(K"	deutscher Zeichensatz
CHR$(27)+"(A"	englischer Zeichensatz
CHR$(27)+"(E"	dänischer Zeichensatz (Nr. 1)
CHR$(27)+"(H"	schwedischer Zeichensatz
CHR$(27)+"(Y"	italienischer Zeichensatz
CHR$(27)+"(Z"	spanischer Zeichensatz
CHR$(27)+"(J"	japanischer Zeichensatz
CHR$(27)+"(6"	norwegischer Zeichensatz
CHR$(27)+"(C"	dänischer Zeichensatz (Nr. 2)
CHR$(27)+"#9"	linken Rand setzen
CHR$(27)+"#0"	rechten Rand setzen
CHR$(27)+"#8"	oberen Rand setzen
CHR$(27)+"#2"	unteren Rand setzen
CHR$(27)+"#3"	Ränder löschen
CHR$(27)+"["+x$+";"+y$+"r"	Seitenkopf x$ Zeilen von oben und Seitenfuß y$ Zeilen von unten
CHR$(27)+"["+x$+";"+y$)+"s"	linken Rand (x$) und rechten Rand (y$) setzen

CHR$(27)+"H"	horizontalen Tabulator setzen
CHR$(27)+"J"	vertikalen Tabulator setzen
CHR$(27)+"[0g"	horizontalen Tabulator löschen
CHR$(27)+"[3g"	alle horizontalen Tabulatoren löschen
CHR$(27)+"[1g"	vertikalen Tabulator löschen
CHR$(27)+"[4g"	alle vertikalen Tabulatoren löschen
CHR$(27)+"#4"	alle Tabulatoren löschen
CHR$(27)+"#5"	Standard-Tabulatoren setzen

Sind bei den Steuersequenzen Werte anzugeben, so sind diese als Zeichenfolge anzugeben (z.B.:x$="123") und nicht etwa als Zeichen mit dem entsprechendem ASCII-Code!

Nun ist es natürlich auch nicht jedermanns Sache, sich eine Steuersequenz wie zum Beispiel 'CHR$(27)"[1m"' zum Umschalten auf Fettschrift auswendig zu merken. Und jedesmal in der Tabelle nachschauen ist vielleicht auch etwas mühsam. Da sollte einmal Nachschauen schon genügen. Aber wozu gibt es schließlich die Möglichkeit, selbstdefinierte Funktionen zu programmieren?:

```
DEFFN fett_ein$ = CHR$(27)+"[1m"
```

Wenn Sie jetzt auf Fettschrift umschalten wollen, genügt ein LPRINT @fett_ein$. Das läßt sich doch schon viel leichter merken! Deshalb mein Vorschlag: Suchen Sie sich aus der Tabelle all die Steuersequenzen heraus, die Sie höchstwahrscheinlich einmal brauchen werden und definieren Sie sie - mit einprägsamen Namen - als Funktionen. Anschließend speichern Sie das Ganze als separates Programm (vielleicht unter dem Namen 'DRUCKERCODES.LST') auf Diskette. Wenn Sie nun in Ihren Programmen auf den Drucker zugreifen wollen, verwenden Sie einfach die gewählten Namen und mergen dann zum Schluß 'DRUCKERCODES.LST' hinten an Ihr Programm.

Die Standard-Steuerzeichen-Tabelle ist zwar eine feine Sache, doch was macht man, wenn der eigene Drucker über Fähigkeiten verfügt, deren Einstellung in der Tabelle überhaupt nicht vorgesehen ist? Man denke nur an Schriftvarianten wie 'Outline',

die mittlerweile bei einigen Druckern zum Standard gehören. In der Steuerzeichen-Tabelle werden Sie diese dagegen vergeblich suchen.

Für solche Fälle verfügt der Amiga über eine zweite Drucker-Schnittstelle: das logische Gerät PAR:. Dieses macht im Grunde genommen nichts anderes als PRT:: Es steuert die parallele Schnittstelle (auch Centronics-Schnittstelle genannt) des Amiga an. Im Gegensatz zu PRT: findet bei PAR: jedoch keine Steuerzeichenübersetzung statt, d.h. die Steuerzeichen werden 'ungefiltert' (unter Umgehung der in den Preferences eingestellten Druckeranpassung) direkt an den angeschlossenen Drucker gesendet. Wie sieht das Ganze nun konkret aus? Zunächst müssen Sie mit

```
OPEN "O",#1,"PAR:"
```

eine logische Datei zu PAR: öffnen. Anschließend senden Sie die erforderliche Steuersequenz mit

```
PRINT #1,CHR$(27).....
```

an den Drucker. Die weiteren Daten (also den auszudruckenden Text) können Sie nun mit

```
PRINT #1,.......
```

ausgeben lassen. Dabei gelten dieselben Syntaxregeln wie bei LPRINT. Zum Schluß vergessen Sie bitte nicht, die Datei mit

```
CLOSE #1
```

wieder zu schließen. Sollte Ihr Drucker zu jenen Exoten gehören, die an der seriellen Schnittstelle des Amiga angeschlossen werden wollen, so hilft Ihnen die dritte Drucker-Schnittstelle des Amiga: das logische Gerät SER:. Die Vorgehensweise ist dabei genau dieselbe wie bei PAR:. Also zuerst mit 'OPEN "O",#1,"SER:" eine logische Datei öffnen, dann mit 'PRINT #1' die Steuerzeichen und sonstigen Daten senden und zum Schluß die Datei mit 'CLOSE #1' wieder schließen.

5.9 Sound- und Spracherzeugung

SOUND { SO } — Ton erzeugen

```
SOUND Frequenz, Dauer [,Lautstärke] [,Kanal]
```

Der Befehl SOUND erzeugt einen Ton, dessen Frequenz zwischen 20 Hertz und 15000 Hertz frei wählbar ist. Der Parameter Dauer ist eine 16-Bit-Zahl, die angibt, wie oft die sog. Hüllkurve des Tones (siehe WAVE) wiederholt wird. Die optional einstellbare Lautstärke kann Werte von 0 (= aus) bis 255 (= volle Lautstärke) annehmen. Die Einteilung ist linear, d.h. ein Wert von 127 ergibt eine mittlere Lautstärke. Dieser Wert ist auch voreingestellt. Mit Kanal läßt sich festlegen, über welchen der vier Tonkanäle des Amiga (Kanal = 0 - 3) der Ton ausgegeben werden soll. Voreingestellt ist der Kanal 0. Jeder Tonkanal läßt sich einzeln ansprechen, d.h. Sie können bis zu vier Töne gleichzeitig spielen. Bei Stereoausgabe stehen die Kanäle 0 und 3 für den linken, die Kanäle 1 und 2 für den rechten Tonausgang.

Neben dem Erzeugen einfacher Töne ist natürlich vor allem eines interessant: das Abspielen von Noten oder ganzen Musikstücken. Dazu muß man die den einzelnen Noten zugeordneten Tonfrequenzen kennen:

Note	Frequenz (Hz)
c	261.6
c#	277.2
d	293.7
d#	311.2
e	329.7
f	349.3
f#	370.0
g	392.0
g#	415.3
a	440.0
a#	466.2
h	493.9

Die Tabelle gibt Ihnen Aufschluß über die Frequenzen der 2. Oktave. Die Frequenzen anderer Oktaven lassen sich daraus denkbar einfach berechnen: Mit jeder Oktave verdoppelt sich die Frequenz einer Note! Die Note a in der 3. Oktave hat also die Frequenz 2*440=880 Hertz, in der 4. Oktave entspricht ihr die Frequenz 2*880=1760 Hertz usw. Für alle, denen das Ausrechnen zu mühsam ist, habe ich die folgende Tabelle zusammengestellt. Sie enthält die gebräuchlichsten Noten und ihre zugehörigen Frequenzen (in Hz):

Oktave:	1	2	3	4	5
c	130.8	261.6	523.2	1046.4	2092.8
c#	138.6	277.2	554.4	1108.8	2217.6
d	146.9	293.7	587.4	1174.8	2349.6
d#	155.6	311.2	622.4	1244.8	2489.6
e	164.9	329.7	659.4	1318.8	2637.6
f	174.7	349.3	698.6	1397.2	2794.4
f#	185.0	370.0	740.0	1480.0	2960.0
g	196.0	392.0	784.0	1568.0	3136.0
g#	207.7	415.3	830.6	1661.2	3322.4
a	220.0	440.0	880.0	1760.0	3520.0
a#	233.1	466.2	932.4	1864.8	3729.6
h	247.0	493.9	987.8	1975.6	3951.2

WAVE { WA } — Hüllkurve festlegen

```
WAVE Kanal, Hüllkurve()
```

Mit WAVE läßt sich für jeden der vier Tonkanäle des Amiga (Kanal = 0 bis 3) eine sog. Hüllkurve festlegen. Diese bestimmt den Klangverlauf des über diesen Kanal ausgegebenen Tones. Die Daten der Hüllkurve werden in dem 256 Elemente umfassenden Integerfeld Hüllkurve() übergeben. Jedes Element des Feldes darf nur Werte zwischen -128 und +127 annehmen. Daher verwendet man am besten ein Byte-Integerfeld (Prefix |).

Wird der Befehl SOUND ohne vorheriges WAVE angewendet, so wird für den angesprochenen Kanal eine Sinusschwingung erzeugt. Diese können Sie auch mit WAVE über das Schlüsselwort SIN() einstellen (WAVE Kanal,SIN()).

SAY — Text sprechen

```
SAY Sprech$ [,Modi%()]
```

Der Befehl SAY dient dazu, einen zuvor mit der Funktion TRANSLATE$ in einen sog. Phonem-Code übersetzten Text zu sprechen. Der Phonem-Code wird dazu in Sprech$ übergeben. SAY und TRANSLATE$() lassen sich auch kombinieren: SAY TRANSLATE$("HALLO").

Durch das sieben Elemente umfassende Feld Modi%() können Sie die 'Stimme' des Amiga auf verschiedene Art und Weise modifizieren:

Modi%(0) Enthält die Frequenz und damit die Höhe der Stimme. Möglich sind Werte von 65 (tiefe Stimme) bis 320 (hohe Stimme). Voreingestellt ist 110.

Modi%(1) Legt die Modulation der Stimme fest. Zur Wahl stehen 'betont' (0) und 'monoton' (1). Voreinstellung: 0.

Modi%(2) Enthält die Sprechgeschwindigkeit in Wörtern pro Minute (40 - 400). Voreingestellt ist 150.

Modi%(3) Legt fest, ob die Stimme männlich (0) oder weiblich (1) klingen soll. Voreingestellt ist 0.

Modi%(4) Enthält die Sampling-Frequenz, die vor allem Auswirkungen auf die Stimmhöhe hat. Möglich sind Werte zwischen 5000 (tief) und 28000 (hoch). Voreinstellung: 22000.

Modi%(5) Legt die Lautstärke der Stimme fest. Erlaubt sind Werte von 0 (nichts zu hören) bis 64 (volle Lautstärke). Voreingestellt ist 64.

Modi%(6) Bestimmt, über welchen der vier Tonkanäle des Amiga (Nummern 0 bis 3) die Stimme zu hören ist. Dabei sind alle Kombinationen möglich:

0 Kanal 0
1 Kanal 1
2 Kanal 2
3 Kanal 3
4 Kanäle 0 und 1
5 Kanäle 0 und 2
6 Kanäle 3 und 1
7 Kanäle 3 und 2
8 Kanäle 0 und/oder 3
9 Kanäle 1 und/oder 2
10 Jedes freie Kanalpaar (Grundeinstellung)
11 Jeder freie Kanal

Die Vielzahl der Parameter ist auf den ersten Blick vielleicht etwas verwirrend. Um die Werte jeweils bequem ändern zu können, ist es am einfachsten, wenn man sich alle Werte für Modi%() in einer DATA-Zeile ablegt und dann dort die erforderlichen Änderungen vornimmt:

```
FOR z%=0 TO 9
  READ Modi%(z%)    !Leseschleife
NEXT z%
!Abgelegte Werte
DATA 110,0,150,0,22000,64,10
```

Um die Stimme zum Beispiel von männlich auf weiblich umzustellen, ändern Sie den vierten Wert auf 1.

Achtung: SAY benötigt für seine Arbeit im DEVS-Ordner der Boot-Diskette das sog. 'Narrator-Device'. Beim Aufruf von SAY sollte sich die Boot-Diskette also in einem der angeschlossenen Laufwerke befinden. Andernfalls erhalten Sie einen System-Requester, der Sie zum Einlegen der Diskette auffordert.

TRANSLATE$() — Text zur Sprachausgabe übersetzen

```
Sprech$=TRANSLATE$("Text")
```

TRANSLATE$ übersetzt einen natürlichsprachlichen Text in einen für den SAY-Befehl verständlichen Phonem-Code. Das Ganze hat allerdings einen Haken: TRANSLATE$ ist im Moment nur in der Lage, englischen Klartext zu bearbeiten, d.h. jeder Text wird nach englischen Ausspracheregeln umgeformt! Deutsche Texte werden dadurch mitunter fast unverständlich.

Eine mögliche Abhilfe besteht darin, den Text so umzuschreiben, daß er - englisch ausgesprochen - deutsch klingt. Zwei Beispiele: Aus einem 'Wie' wird ein 'Vee', ein 'sch' wird zu 'sh'. Leider lassen sich dabei aber keine allgemeingültigen Umformungsregeln aufstellen. Im Einzelfall hilft nur Ausprobieren.

6. Programmstruktur

6.1 Schleifenkonstruktionen

DO ... LOOP { DO ... L }

```
Endlosschleife DO [WHILE Bed] [UNTIL Bed] ...Doppelt bedingte Schleife
LOOP [WHILE Bed] [UNTIL Bed]

DO
...   auszuführende Programmteile
LOOP
oder:
DO [WHILE Bedingung] [UNTIL Bedingung]
...   auszuführende Programmteile, wenn DO-Bedingung
...   wahr ist bzw. solange LOOP-Bedingung wahr ist.
LOOP [WHILE Bedingung] [UNTIL Bedingung]
```

Eine DO...LOOP-Schleife bricht im Normalfall nur dann ab, wenn sie auf eine Abbruch-Anweisung (END, EDIT, STOP) trifft, eine EXIT IF-Anweisung findet und die Abbruchbedingung wahr ist, eine GOTO-Anweisung innerhalb der Schleife zu einem Label außerhalb der Schleife verzweigt oder die Break-Funktion verwendet wird.

Außerdem ist es möglich, eine DO...LOOP-Konstruktion mit Ein- und Ausgangsbedingungen zu versehen. Dazu kann sowohl bei DO als auch bei LOOP entweder eine WHILE- oder eine UNTIL-Bedingungsabfrage hinzugefügt werden (siehe WHILE... WEND bzw. REPEAT...UNTIL).

Wenn allein DO...LOOP eingesetzt wird, ist es möglich, statt LOOP auch ENDDO zu verwenden (wird vom Interpreter dann durch LOOP ersetzt).

DO...LOOP kann - wie alle anderen Schleifen auch - beliebig tief verschachtelt werden. Beachten Sie bitte das Beispiel zu EXIT IF.

FOR ... NEXT { F ... N } Zählschleife

```
FOR Zaehl=Start TO [DOWNTO] Ende [STEP Schritt]
...  auszuführende Programmteile
NEXT Zaehl
```

Mit der Konstruktion FOR...NEXT wird eine indizierte Wiederholungschleife angelegt.

Die Kopfzeile der Schleife enthält den Anfangswert Start und den Endwert Ende. Die numerische Zählvariable Zaehl wird bei Schleifenbeginn mit dem Wert Start belegt und dann bei jedem Durchlauf solange erhöht bzw. vermindert, bis sie den Wert Ende erreicht hat. Dabei werden alle Programmzeilen, die zwischen FOR und dem dazugehörigen NEXT eingeschlossen sind, bei jedem Durchlauf ausgeführt. Nach Erreichen des Ende-Wertes wird das Programm mit der auf die NEXT-Anweisung folgenden Programmzeile fortgesetzt.

Wird nur FOR..TO..NEXT ohne die Option STEP verwendet, beträgt die Schrittweite immer +1. Bei Verwendung von FOR..DOWNTO..NEXT ist der Anfangswert größer als der Endwert anzugeben, da in diesem Fall die Schrittweite immer -1 beträgt. Die Verwendung von STEP ist hier nicht zulässig. Die Option STEP (nur bei TO-Schleifen) bewirkt, daß der nach STEP angegebene Wert oder Ausdruck als Schrittweite angenommen wird. Hier sind auch negative Werte möglich, wobei dann sinnvollerweise - wie bei DOWNTO - der Endwert kleiner als der Startwert zu wählen ist.

Ist bei Verwendung von STEP der Wertebereich von Start bis Ende nicht glatt durch den STEP-Wert Schritt teilbar, errechnet sich der zuletzt verarbeitete Zählwert aus:

```
Abs(Ende-Start)-((Abs(Ende-Start)) Mod Abs(Schritt))
```

Ist bei positiven Schleifen Start größer als Ende bzw. bei negativen Schleifen Ende größer als Start, so wird die Schleife trotzdem mindestens einmal mit dem Start-Wert durchlaufen.

Statt NEXT Var kann auch ENDFOR Var angegeben werden.

REPEAT ... UNTIL { REP ... U } — Bedingte Schleife

```
REPEAT
... auszuführende Programmteile
UNTIL Bedingung
```

Die REPEAT...UNTIL-Schleife kann immer dann angewendet werden, wenn die Anzahl der Schleifendurchläufe nicht durch das Erreichen eines Endwertes (FOR...NEXT) festgelegt ist und die Schleife mindestens einmal durchlaufen werden soll.

Die Bedingung zum Verlassen der Schleife wird hier am Schleifenende geprüft. D.h., daß die Schleife mindestens einmal bis zu der in UNTIL vereinbarten Bedingung durchlaufen wird. Es sei denn, daß innerhalb der Schleife eine EXIT IF-Bedingung mit "wahr" beantwortet wird (siehe EXIT IF). Ist "Bedingung" wahr, wird das Programm mit der nächsten auf UNTIL folgenden Zeile fortgesetzt.

Diese Konstruktion läßt sich bestens dazu verwenden, um mehrfach kombinierte Abbruchbedingungen zu stellen. So können z.B. gleichzeitig ein bestimmter Tastatur-, Maus- und/oder Mausknopf-Status, der Inhalt von Variablen und/oder das Erreichen bestimmter Limits als Bedingung(en) angegeben werden.

Angenommen, für den Abbruch einer REPEAT...UNTIL-Schleife sollen die folgenden Bedingungen ausschlaggebend sein:

Mausklick rechts und Zählwert größer 100,

oder

<Esc>-Taste gedrückt,

oder es sind mehr als 10 Sekunden vergangen. Die Schleifenkonstruktion sieht dann so aus:

```
T%=Timer                        ! Timer festhalten
Repeat                          ! Schleifenstart
  Inc A%                        ! Zähler +1
  K=Mousek                      ! Maustasten-Abfrage
  Key%=Asc(Right$(Inkey$))      ! Tastatur-Abfrage
Until (K=2 And A%>100) Or Key%=27 Or (Timer-T%)>2000
```

Mit den Booleschen Operatoren AND/OR/NOT/XOR/IMP/EQV lassen sich diese Bedingungen auf das Abenteuerlichste miteinander verknüpfen. Statt UNTIL kann auch ENDREPEAT verwendet werden. Der Interpreter wandelt diesen Ausdruck dann selbständig in UNTIL um.

WHILE ... WEND { W ... WE } — Bedingte Schleife

```
WHILE Bedingung
...  auszuführende Programmteile ...
WEND
```

Die WHILE...WEND-Schleife hat die Eigenschaft, daß die Abfrage der Lauf-Bedingung bereits am Anfang der Schleife stattfindet.

So kann es sein, daß die Schleife zwar in der ersten Zeile betreten wird, jedoch das Programm sofort hinter dem dazugehörigen WEND fortgesetzt wird. Dann nämlich, wenn die Bedingung bereits bei Betreten der Schleife nicht erfüllt ist. In diesem Fall werden die zwischen WHILE und WEND eingeschlossenen Programmzeilen also nicht ausgeführt. Beachten Sie hierzu bitte das Beipiel zu EOF().

Statt WEND kann auch ENDWHILE verwendet werden. Der Interpreter wandelt diesen Ausdruck dann selbständig in WEND um.

6.2 Bedingte Verzweigungen

EXIT IF { EX } — Bedingter Schleifenabbruch

```
EXIT IF Bedingung
```

Das ist ein sehr nützlicher Befehl, wenn es darum geht, innerhalb von FOR...NEXT-, DO...LOOP-, REPEAT...UNTIL- oder WHILE...WEND-Schleifen an beliebigen - und beliebig vielen - Stellen zusätzliche Abbruchbedingungen stellen zu können.

Eine solche Schleife kann unabhängig von ihrem Zustand jederzeit verlassen werden, sobald darin eine durch EXIT IF gestellte Bedingung mit "wahr" beantwortet wird. Das Programm wird dann mit der auf den zugehörigen Schleifenwendepunkt folgenden Programmzeile fortgesetzt. Dabei ist zu beachten, daß das Programm direkt vom auslösenden EXIT IF hinter den Wendepunkt springt. Zwischen dem auslösenden EXIT IF und dem Wendepunkt stehende Programmzeilen werden also nicht noch einmal ausgeführt.

Das Beispiel zu REPEAT...UNTIL läßt sich nun anhand einer DO...LOOP-Schleife und EXIT IF folgendermaßen umstrukturieren:

```
T%=Timer                        ! Timer festhalten
Do                              ! Schleifenstart
  Exit If (Timer-T%)>2000       ! 10 Sekunden vergangen?
  Inc A%                        ! Zähler +1
  K=Mousek                      ! Maustasten-Abfrage
  Key%=Asc(Right$(Inkey$))      ! Tastatur-Abfrage
  Exit If Key%=27               ! <Esc>-Taste gedrückt?
  Exit If K=2 And A%>100        ! Mausklick rechts und
  '                             ! Zähler größer 100?
Loop
```

Die - oben schon erwähnte - Besonderheit ist hier, daß die Schleife direkt verlassen wird, sobald 10 Sekunden vergangen sind und bis dahin keine der beiden anderen Abbruchbedingungen erfüllt wurde. In diesem Fall heißt das, daß der vor dem auslösenden Timer-Abbruch gültige Maustasten- und Tastaturstatus sowie der Zähler A% erhalten bleiben und ggf. weiterverwertet werden können.

EXIT IF kann auch innerhalb von IF...ENDIF- oder SELECT...ENDESELECT-Blöcken eingesetzt werden. Das Programm wird dann ggf. hinter der Schleife fortgesetzt, in der diese Befehle stehen.

IF [ELSE] ENDIF { I... [E...] EN } — Bedingungsabfrage

ELSE IF { E } — Unter-Bedingungsabfrage

```
IF Bedingung [THEN]
...  auszuführende Programmteile,
     wenn Bedingung wahr ist
[ELSE
...  auszuführende Programmteile,
     wenn Bedingung unwahr ist ]
ENDIF
oder:
IF Bedingung1 [THEN]
...  auszuführende Programmteile,
     wenn Bedingung1 wahr ist
[ELSE IF Bedingung2
...  auszuführende Programmteile, wenn Bedingung1
     unwahr und Bedingung2 wahr ist.]
[ELSE IF Bedingung3
...  auszuführende Programmteile, wenn alle
     vorherigen Bedingungen unwahr waren,
     jedoch Bedingung3 wahr ist.]
...
...  ggf. weitere ELSE IF-Abfragen
```

```
[ELSE
...  auszuführende Programmteile, wenn alle
     vorherigen Bedingungen unwahr waren.]
ENDIF
```

In GFA-BASIC ist es möglich, die Ausführung auch umfangreicher Programmteile allein von der Erfüllung einer einzigen Bedingung abhängig zu machen. Die Folge an Befehlen, die abhängig von einer IF-Bedingung ausgeführt werden sollen, wird hier nur von dem zugehörigen ENDIF (bzw. ELSE) eingegrenzt.

Ist die Bedingung wahr, werden die zwischen IF und dem zugehörigen ENDIF bzw. ELSE stehenden Programmteile ausgeführt. Bei Verwendung der Option ELSE werden die zwischen ELSE und dem zugehörigen ENDIF eingeschlossenen Befehle ausgeführt, wenn die Bedingung sich als unwahr erweist. Ist ein IF- oder ELSE-Block ausgeführt, wird das Programm mit der auf das zugehörige ENDIF folgenden Programmzeile fortgesetzt.

IF-Abfragen können beliebig tief verschachtelt werden. Der optionale Zusatz THEN hinter IF ist zur Kompatibilität mit anderen BASIC-Dialekten gedacht. Er kann in GFA-BASIC vernachlässigt werden.

Durch den Zusatzbefehl ELSE IF ist es möglich, sich Verschachtelungen folgender Art zu ersparen:

```
If Bedingung1
... Programmblock 1    >--------.
Else                            |
  If Bedingung2                 |
  ... Programmblock 2  >------. |
  Else                        | |
    If Bedingung3             | |
    ... Programmblock 3 >---. | |
    Else                    | |
    ... Programmblock 4 >-. | |
    Endif                 | | |
  .<----------------------'-' |
  Endif                       | |
.<----------------------------' |
Endif                           |
.<------------------------------'
```

Mit ELSE IF sieht dieselbe Struktur so aus:

```
If Bedingung1
... Programmblock 1  >--------.
Else If Bedingung2            |
... Programmblock 2  >------. |
Else If Bedingung3          | |
... Programmblock 3  >----. | |
Else                      | | |
... Programmblock 4  >--. | | |
Endif                   | | | |
.<----------------------'-'-'-'
```

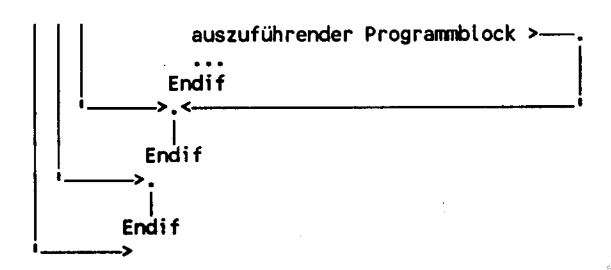
Ist hier die Eingangs-IF-Bedingung unwahr und trifft das Programm auf eine ELSE IF-Abfrage, deren Bedingung wahr ist, wird nur der darunter angegebene Programmblock abgearbeitet und nach dessen Ausführung zu der auf das zugehörige ENDIF folgenden Programmzeile gesprungen. Wird als Abschluß die Option ELSE verwendet und keine der vorangegangenen ELSE IF-Bedingungen wurde mit wahr beantwortet, wird auch hier die unter ELSE angegebene Programmfolge alternativ zu allen Vorbedingungen ausgeführt.

Bei folgender Abfrage werden mehrere Bedingungen durch AND so miteinander verknüpft, daß der dahinterstehende Programmblock nur dann ausgeführt wird, wenn alle Bedingungen der Abfrage erfüllt sind. Statt des oben verwendeten Ausdrucks Bedingung setze ich hier den Ausdruck Case ein, der jedoch nichts mit der CASE-Abfrage zu tun hat.

```
If Case1=True And Case2=True And Case3=False
  ...
  auszuführender Programmblock
  ...
Endif
```

Eine solche Struktur ließe sich mit derselben Wirkung auch in mehrere IF-Abfragen auflösen:

```
.---      If Case1=True
|          '---------------.
| .---      If Case2=True <---'
| |          '---------------.
| | .---      If Case3=False <---'
| | |          ...
```



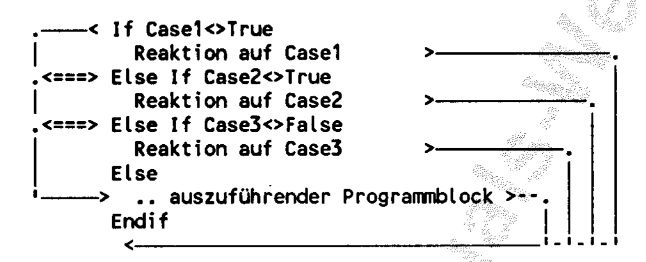
Auch hier wird der Block nur ausgeführt, wenn alle Abfragen wahr sind. Auf den ersten Blick mag die zweite Version etwas aufwendiger erscheinen. Sie hat jedoch den Vorteil, daß nach jeder Einzel-Abfrage auf die Erfüllung der einzelnen Bedingung reagiert werden kann.

```
Fileselect "auswahl","laden","df0:",F$
If F$>""                                    ! Case1
  Print "Okay-Box oder Doppelklick"         ! Reaktion
  If Right$(F$,4)=".DAT"                    ! Case2
    Print "Korrekte Extension               ! Reaktion
    If Len(F$)>4                            ! Case3
      Print "Dateiname ";Left$(F$,Len(F$)-4) ! Block..
      Print "Extension = .DAT"              ! ...ausführen
    Endif
  Endif
Endif
```

Ähnliches läßt sich auch mit mehreren ELSE IFs anstatt einer OR-Kette realisieren. Bei der OR-Kette wird die Abfrage passiert und der Programmblock ausgeführt, sobald nur eine Bedingung der Kette wahr ist. Hier wurden - um einen Vergleich mit der oben beschriebenen AND-Kette zu ermöglichen - die Bedingungen negiert.

```
If Case1<>True Or Case2<>True Or Case3<>False
  Reaktion auf Negation der Bedingungen
Else
  ...
  auszuführender Programmblock
  ...
Endif
```

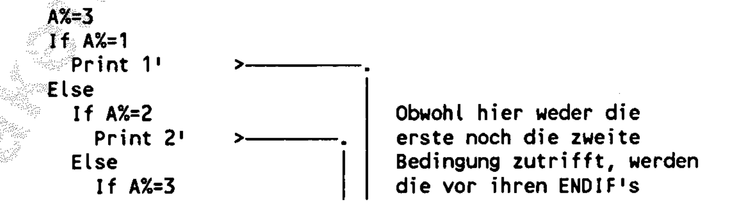
Mit der ELSE IF-Struktur kann jede der drei OR-Bedingungen einzeln abgefragt werden und dann auf die Erfüllung der jeweiligen Bedingung separat reagiert werden:

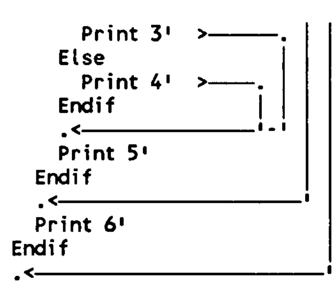


In der Praxis könnte das dann so aussehen:

```
Fileselect "auswahl","laden","df0:",F$
If F$=""                               ! Case1
  Print "Abbruch-Box angeklickt"       ! Reaktion
Else If Right$(F$,4)<>".DAT"           ! Case2
  Print "Falsche Extension (kein .DAT) ! Reaktion
Else If Len(F$)=4                      ! Case3
  Print "Kein vollständiger Dateiname " ! Reaktion
Else                                   ! Keine der Bedingungen ist
                                         wahr!
  Print "Korrekte Dateiauswahl!"       ! Block...
Endif                                  !...ausführen
```

Der wesentliche Unterschied zwischen der Normal-IF-Struktur und der ELSE IF-Struktur ist, daß beim Normal-IF nach jedem Programmblock die ENDIFs der vorangegangenen Abfragen noch passiert werden. Daraus ergibt sich die Möglichkeit, abhängig von dem ausgeführten Block weitere Programmteile ausführen zu lassen, die mit den vorangegangenen Abfragen in Bezug stehen. Die Notwendigkeit zu Raffinessen dieser Art ist zwar äußerst gering, kann jedoch nicht absolut ausgeschlossen werden. Beispiel:



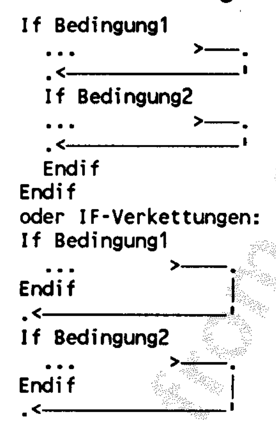


stehenden Zeilen Print 5 und Print 6 ausgeführt. Wäre A% = 2, würde die Zeile Print 5 nicht mehr ausgeführt werden. Bei A% = 0 würden außer der letzten ELSE-Anweisung Print 4 ebenfalls noch Print 5 und Print 6 ausgeführt werden.

Diese Möglichkeit ist bei der ELSE IF-Struktur nicht gegeben, da dort einfach kein Platz für die Zeilen Print 5 und Print 6 wäre.

Gemeinsam ist beiden Strukturen, daß immer dann, wenn eine Bedingung zutrifft, die nachfolgenden Abfragen übersprungen, also ausgeschlossen werden.

Bei IF-Verschachtelungen



ist dieser Ausschluß nicht gewährleistet, da mit einem Fall auch mehrere Bedingungen gleichzeitig erfüllt sein können.

Sie werden sich evtl. fragen, wie denn überhaupt die Entscheidung wahr oder unwahr getroffen wird. Bei einfachen Abfragen, wie z.B. If A%=1, ist das leicht zu erklären. Hat die Variable

A% den Wert 1, wird die Bedingung mit wahr (TRUE = -1) beantwortet, andernfalls mit unwahr (FALSE = 0). Geben Sie bitte im Direktmodus ein:

```
Print 1=1
```

Der Interpreter vergleicht nach dem Booleschen Verfahren "logisch" die beiden gegenübergestellten Werte 1 und 1. Anschließend liefert er Ihnen den Wahrheitswert -1 (TRUE). Wie das Ganze maschinenintern durch sogenannte Gatter geregelt wird, soll uns hier nicht interessieren. Wichtig ist, daß Sie wissen, daß es so ist. Schauen Sie sich bitte dazu die Bedeutung der einzelnen Operatoren in Kapitel 8.1 sowie die Vorrang-Regelung der Operatoren untereinander an.

Bei komplizierteren Verknüpfungen gibt es schon einige Probleme. Hier ist es wichtig, auf eine exakte und logisch richtige Klammersetzung zu achten. So können sich aus (oberflächlich gesehen) immer demselben Ausdruck mehrere Resultate ergeben:

```
Print (2^2=4)+((7-1)>3)   Ausgabe: -2
Print 2^2=4+((7-1)>3)     Ausgabe:  0
Print (2^2=4)+(7-1)>3     Ausgabe: -1
Print (2^2=4)+7-(1>3)     Ausgabe:  6
```

Verblüffend, oder? - Die Klammersetzung macht es möglich! Nun die Erklärung: In der ersten Zeile wird der Ausdruck (2^2=4) zusammengefaßt. Dasselbe geschieht mit ((7-1)>3). Beide Ausdrücke werden - jeder für sich - auf ihren Wahrheitsgehalt geprüft. Der erste Ausdruck ergibt TRUE (-1), da die zweite Potenz von 2 tatsächlich 4 ist. Der zweite Ausdruck liefert ebenfalls TRUE (-1), da 7 minus 1 (also 6) tatsächlich größer als 3 ist. Durch das Plus-Zeichen werden nun beide Wahrheitswerte addiert. Und das ergibt (-1)+(-1)= -2.

In der zweiten Zeile habe ich einfach die Klammer um den ersten Ausdruck weggelassen. Daraus ergibt sich eine völlig andere Konstellation. In diesem Fall wird zuerst das Ergebnis von 2^2 berechnet (4). Das Gleichheitszeichen wird hier allerdings nicht als zum ersten Ausdruck zugehörig erkannt, sondern stellt den

Verknüpfungs-Operator zum zweiten Ausdruck dar. Dieser liefert - wie in der ersten Zeile - wieder TRUE (-1). In der endgültigen Auswertung ergibt sich also 4=(-1). Da 4 nun aber nicht -1 ist, liefert die Zeile den Unwahrheitswert FALSE, also 0.

In der dritten Zeile haben wir als ersten Ausdruck wieder (2^2=4). Hier ist die Änderung beim zweiten Ausdruck zu finden. Es fehlt die äußere Klammer. 2 hoch 2 ist 4 - stimmt! Also erhalten wir aus dem ersten Ausdruck wieder TRUE (-1). Der zweite Ausdruck ist in diesem Fall jedoch auf (7-1) reduziert. Das ergibt 6. Als nächstes werden die beiden Ausdrücke addiert. 6+(-1) ist 5. Dieses Additionsergebnis wird nun abschließend daraufhin geprüft, ob es größer als 3 ist. Da 5 tatsächlich größer als 3 ist, erhalten wir als Gesamtergebnis dieser Zeile ein TRUE (-1).

Die vierte und letzte Zeile hat immer noch - oberflächlich gesehen - eine frappierende Ähnlichkeit mit ihren Vorgängerinnen. Hier habe ich einfach die 7 aus dem zweiten Ausdruck ausgeklammert und (1>3) zusammengefaßt. Der erste Ausdruck ist uns schon bekannt. Er liefert - wie in Zeile 1 und Zeile 3 - den Wert -1 (TRUE). Da die Klammersetzung absoluten Vorrang vor allen anderen Operatoren hat, berechnen wir als nächstes den Ausdruck (1>3). Sehr wahrscheinlich wissen Sie so gut wie ich, daß 1 nicht größer als 3 ist. Also erhalten wir den Wert 0 (FALSE). Mit etwas mathematischer Vorkenntnis lassen sich die drei Faktoren nun folgendermaßen zusammenfassen: (-1)+7-(0) = 6.

So weit, so gut - wenn da nicht noch die logischen Operatoren wären.

```
Print 10*10 And 14 Or 3< >2      Ausgabe:  -1
Print 10*10 And (14 Or 3)< >2    Ausgabe:  100
Print 10*(10 And 14 Or 3< >2)    Ausgabe:  -10
```

Drei neue Zeilen, die - für den Laien - fast identisch sind, und trotzdem verschiedene Ergebnisse liefern.

Des Rätsels Lösung: In Zeile 1 finden Sie keine Klammer. Daher eignet sie sich besonders dazu, die Prioritäten der verwendeten Operatoren zu erläutern. Als erstes wird in diesem Fall die Punkt-Rechnung durchgeführt, also 10*10. Das ergibt 100. Als nächstes kommt die Vergleichsoperation <> dran. Alle Vergleiche haben Vorrang vor logischen Operatoren. Es soll hier festgestellt werden, ob 3 ungleich 2 ist. Da es das ist, erhalten wir hieraus den Wert -1 (TRUE). Zum Schluß bearbeiten wir in dieser Zeile die Booleschen Operatoren. Da beide auf derselben Ebene liegen, ergibt sich die Frage, welche Operation von beiden zuerst dran kommt - AND oder OR? In solchen Fällen gilt die goldenen Regel: immer von links nach rechts! Ich setze nun voraus, daß Sie sich über das Verfahren der logischen Verknüpfungen im Kapitel 4 "Basis-BASIC" eingehend informiert haben.

```
        100 -> Binär:    01100100
AND      14 -> Binär:    00001110
                         ________
                         00000100  -> Dezimal: 4  >--.
.----------------------------------------------------'
'->       4 -> Binär:         100
OR       -1 -> Binär:  1...111111
                       __________
                       1...111111  -> Dezimal: -1
                                      ============
```

Als Ergebnis der ersten Zeile erhalten wir den Wert -1 (TRUE). In Zeile 2 habe ich hier nichts weiter gemacht, als (14 Or 3) zu einem Ausdruck zusammenzufassen. Da - wie gesagt - die Klammerrechnung Vorrang hat, erhalten wir daraus:

```
         14 -> Binär:    00001110
OR        3 -> Binär:    00000011
                         ________
                             1111  -> Dezimal: 15
                                      ============
```

Wie in Zeile 1 erhalten wir auch hier aus 10*10 den Wert 100. Weil auch hier wieder der Vergleich <> Vorrang vor AND hat, beziehen wir nun daraus den Wert -1 (TRUE), da der Klammerwert 15 tatsächlich ungleich 2 ist. Zum Schluß verknüpfen wir die beiden Teilergebnisse im AND-Modus:

```
         100 -> Binär:   01100100
AND       -1 -> Binär: 1..1111111
                       ----------
                         01100100  -> Dezimal: 100
                                      ==============
```

Das Ergebnis der zweiten Zeile heißt 100.

Die dritte Zeile ist wiederum nur geringfügig variiert. Wie gesagt, es wird erst einmal die Klammer berechnet. Darin finden wir als höchste Priorität den Ungleich-Vergleich <>. Ist drei ungleich zwei? Ja, also erhalten wir TRUE (-1). Als zweites stehen sich eine AND- und eine OR-Verknüpfung gleichwertig gegenüber. Ich fange links an:

```
          10 -> Binär:   00001010
AND       14 -> Binär:   00001110
                         --------
                         00001010  -> Dezimal: 10  >-.
 .---------------------------------------------------'
 '->      10 -> Binär:       1010
OR        -1 -> Binär: 1...111111
                       ----------
                       1...111111  -> Dezimal: -1
                                          ========
```

Als Ergebnis des Klammerausdrucks erhalten wir wieder -1 (TRUE). Diese -1 wird nun mit dem Wert 10 multipliziert, und siehe da - es ergibt sich der Negativ-Wert -10.

Ich hoffe, daß es mir anhand dieser zwei kleinen Beispiele gelungen ist, Ihr kritisches Auge im Umgang mit Verknüpfungen dieser Art etwas geschärft zu haben. Nun wissen Sie wenigstens ungefähr, was Ihnen an Verwirrungen bevorstehen kann, und über Bedingungen wie die folgende wundern Sie sich dann hoffentlich nicht mehr all zu sehr.

```
If (10*(10 And 14 Or 3<>2))=-10
  Print "Heureka!"
Endif
```

Als Leckerbissen folgt nun ein kleines Programm, das Ihnen als Anregung für die Lösung sogenannter Kniffel-Aufgaben dienen soll.

Stellen Sie sich bitte vor, Sie hätten folgende Aufgabe zu lösen:

XX * X	Eine zweistellige Zahl ist mit einer einstelligen Zahl zu multiplizieren.
XX	Das Ergebnis muß wieder zweistellig sein.
+ XX	Dazu wird ein zweistelliger Wert addiert, und das Ergebnis der Rechnung muß wieder
= XX	zweistellig sein.

Der Witz an der Sache ist nun, daß für alle neun möglichen Stellen alle neun Ziffern von 1 bis 9 je einmal verwendet werden müssen. Es darf also jede Ziffer nur einmal vorkommen. Es ist nur ein einziges korrektes Ergebnis möglich.

Das nun folgende Programm löst diese Aufgabe völlig "unmathematisch", und es sind natürlich wesentlich elegantere mathematische Lösungen vorstellbar. Da diese jedoch extrem kompliziert sind und von Ihnen nicht erwartet werden kann, sich in die höhere Mathematik zu vertiefen, habe ich die "logische" Lösung gewählt. Mathematiker mögen mir verzeihen. Ich habe es mit der Algebra versucht, mußte jedoch nach mehreren erfolglosen Ansätzen aufgeben.

Ein weiterer Effekt dieses Programms ist der, daß daran hervorragend der zeitliche Aufwand verschiedener Konstellationen nachvollzogen werden kann. Versuchen Sie dazu einmal, das Programm so zu variieren, daß trotz der Änderung der Algorithmus logisch erhalten bleibt. Beispielsweise kann die vorletzte IF-Abfrage

```
If Len(Ergebnis$+B$)=9
```

ohne weiteres um drei Zeilen - direkt unter B$=Str$(B%) - versetzt werden. Außerdem könnte man die String-Umwandlung I$=Str$(I%) streichen und den Ausdruck Str$(I%) statt I$ direkt einsetzen. Oder Sie verwenden statt der 4-Byte-Integer I%, J%, K%, A%, B% und C% einfach Realvariablen. Der Zeitaufwand wird sich in allen drei Fällen erhöhen. In der Wahl der Varianten lassen Sie bitte Ihre Phantasie spielen. Das Wissen um den

jeweiligen Zeitaufwand wird Ihnen später bei der Entwicklung und Optimierung eigener Programme sehr von Nutzen sein.

```
T%=Timer                           ! Start-Timer festhalten
For I%=12 To 98                    ! Zählschleife für ersten
 '                                 ! zweistelligen Wert
 If (I% Mod 11)>0 And (I% Mod 10)>0 ! Ist I% durch 11 oder 10
  ' glatt teilbar, so kann es sich nur um Schnapszahlen
  ' (22, 33 etc.) oder Nullzahlen (20, 30 etc.) handeln und
  ' die können wir hier nicht gebrauchen.
  I$=Str$(I%)                      ! I% in String umwandeln
  I2$=Str$(I% Mod 10)              ! Einerstelle von I%
  I3$=Str$(I% Div 10)              ! Zehnerstelle von I%
  For J%=1 To 9                    ! Einstellige Schleife
   J$=Str$(J%)                     ! J% in String umwandeln
   If Instr(I$,J$)=0               ! Ist Ziffer J% in den
    '                              ! I%-Ziffern enthalten?
    A%=I%*J%                       ! Nein, dann multiplizieren.
    If A%>11 And A%<99             ! Ergebnis zweistellig und
     If (A% Mod 11>0) And (A% Mod 10>0) ! glatt durch 11 oder
      '                            ! 10 teilbar? (s.o.)
      Clr C%                       ! INSTR-Positionsspeicher klar
      A$=Str$(A%)                  ! A% in String umwandeln
      A2$=Str$(A% Mod 10)          ! Einerstelle von A%
      A3$=Str$(A% Div 10)          ! Zehnerstelle von A%
      C%=Instr(A$,I3$)             ! Zehnerstelle von I% in A%?
      C%=Max(C%,Instr(A$,I2$))     ! Einerstelle von I% in A%?
      C%=Max(C%,Instr(A$,J$))      ! J% in A% enthalten?
      If Len(I$+J$+A$)=5 And C%=0  ! Ziffernanzahl der
       ' bisherigen Rechnung = 5 und keine der bisher
       ' verwendeten Ziffern in einer anderen enthalten?
       For K%=12 To 98             ! Schleife für die zu
        '                          ! addierende Zahl
        K$=Str$(K%)                ! K% in String umwandeln
        K2$=Str$(K% Mod 10)        ! Einerstelle von K%
        K3$=Str$(K% Div 10)        ! Zehnerstelle von K%
        Ergebnis$=I$+J$+A$+K$      ! Ziffernanzahl der bisherigen
        '                          ! Rechnung feststellen
        If Len(Ergebnis$)=7        ! Anzahl = 7?
         If (K% Mod 11>0) And (K% Mod 10>0) ! ist K%
          ' durch 10 oder 11 glatt teilbar? (s.o.)
          C%=Instr(A$,K3$)         ! Zehnerstelle von K% in A%?
          C%=Max(C%,Instr(A$,K2$)) ! Einer von K% in A%?
          C%=Max(C%,Instr(I$,K3$)) ! Zehner von K% in I%?
          C%=Max(C%,Instr(I$,K2$)) ! Einer von K% in I%?
          C%=Max(C%,Instr(K$,J$))  ! J% in K% enthalten?
          If C%=0                  ! Keine der bisherigen
           ' Ziffern in einer anderen Zahl enthalten?
           B%=A%+K%                ! Multiplikationsergebnis
           '                       ! und K% addieren
           B$=Str$(B%)             ! B% in String umwandeln
           If B%>11 And B%<99      ! Ergebnis zweistellig?
```

```
              If (B% Mod 11>0) And (B% Mod 10>0) ! Ist es
               ' weder durch 10 noch 11 teilbar? (s.o.)
               If Len(Ergebnis$+B$)=9 ! Anzahl aller
                ' verwendeten Ziffern = 9?
                C%=Instr(B$,I3$) !————————. Prüfen, ob
                C%=Max(C%,Instr(B$,I2$))   ! eine der
                C%=Max(C%,Instr(B$,A3$))   ! bisherigen
                C%=Max(C%,Instr(B$,A2$))   ! Ziffern im
                C%=Max(C%,Instr(B$,K3$))   ! Endergebnis
                C%=Max(C%,Instr(B$,K2$))   ! enthalten
                C%=Max(C%,Instr(B$,J$)) !--' sind
                If C%=0                   ! Nein?
                 Print "Sek. : ";(Timer-T%)/200
                 Print
                 Print ''I%;" * ";J%      ! dann
                 Print "  ————————"       ! Lösung
                 Print "     ";A%         ! ausgeben
                 Print "  + ";K%          ! ...
                 Print "  ————————"       ! ...
                 Print "  = ";B%          ! ...
                 End                      ! und Ende !!
                Endif
               Endif     ! Für die, die nichts mit der Funktion
              Endif      ! MAX() in den INSTR-Abfragen anfangen
             Endif       ! können:
            Endif        ! Durch INSTR soll festgestellt werden,
           Endif         ! ob eine der jeweils verwendeten Ziffern
          Endif          ! schon vorher verwendet wurde.
         Next K%         ! Da mehrere INSTR-Abfragen hintereinander
        Endif            ! stehen und anschließend auf Null getestet
       Endif             ! wird, kann eine hintenstehende Abfrage den
      Endif              ! C%-Wert einer vorangegangenen mit Null
     Endif               ! überschreiben. Um evtl. schon gefundene
    Next J%              ! Positionen zu erhalten, wird sie durch
   Endif                 ! MAX() in die jeweils nächste INSTR-Abfrage
  Next I%                ! hinübergerettet.
```

SELECT [CONT] CASE [TO] [DEFAULT] ENDSELECT

{ S }{ CON }{ CA }{ DEFA }{ ENDS }
Fall-Entscheidung

```
SELECT Expr
CASE Konstante1 [TO Konstante2 [,[...] TO [...]]]
  ...  auszuführende Programmteile, wenn Expr gleich
       Konstante1, bzw. - bei Option TO - wenn Expr
       innerhalb des Bereichs von Konstante1 bis
```

```
      Konstante2 liegt.
  [CONT]
[CASE Konstante1 [,Konstante2 [,Konstante3 [,...]]]
  ... auszuführende Programmteile, wenn Expr gleich
      Konstante1 oder gleich Konstante2 oder
      gleich Konstante3 oder ... oder ...]
  ... ggf. weitere CASE-Entscheidungen
  [CONT]
[DEFAULT
  ... auszuführende Programmteile, wenn keine der
      vorhergehenden Abfragen zugetroffen hat.]
ENDSELECT
```

Diese Fallentscheidung bietet die Möglichkeit zu einer Expr-abhängigen Verzweigung (select = auswählen/case = falls). Viele werden diese Konstruktion aus der Sprache C kennen (Switch/Case). Expr kann ein beliebiger numerischer oder alphanumerischer Ausdruck sein, dessen Ergebnis ggf. vorher ermittelt wird. Es ist auch die Angabe von Variablen oder Konstanten möglich.

Wird ein(e) alphanumerischer Ausdruck, Konstante oder Variable in Expr verwendet, werden davon nur die ersten vier Zeichen zum Vergleich herangezogen. Diese werden dann intern in einen 4-Byte-Wert umgewandelt.

Hinter CASE wird bei Werte-SELECT in Konstante ein numerischer Wert oder ein max. vier Zeichen langer Text als Konstante oder String-Variable angegeben, der dann daraufhin überprüft wird, ob Expr ihm entspricht. Bei Werte-SELECT angegebene Strings werden auf Gültigkeit geprüft, indem der SELECT-Wert mit den ASCII-Werten der ersten vier Zeichen des Textes (sofern vorhanden) verglichen wird. Beispiel:

```
A%=ASC(inkey$)*2^24+ASC(inkey$)*2^16+ASC(inkey$)*2^8+ASC(inkey$)
SELECT A%
CASE "abcd"
  PRINT "abcd wurde eingegeben!"
DEFAULT
  PRINT "irgendwas!"
ENDSELECT
```

In diesem Fall werden vier Tasten abgefragt. Der ASCII-Wert aller einzelnen Tasten wird zu einen Gesamt-4-Byte-Wert ver-

knüpft. Auf dieselbe Art analysiert CASE bei Werte-SELECT einen angegebenen String. Es wird also ein MKL$-, MKI$- oder CHR$-String gebildet (je nach Länge des angegebenen CASE-Strings), der dann mit Expr verglichen wird.

3-Zeichen-Strings:

```
  (ASCII-Wert des 1. Zeichens) * (2^16)
+ (ASCII-Wert des 2. Zeichens) * (2^8)
+ (ASCII-Wert des 3. Zeichens)
```

2-Zeichen-Strings:

```
  (ASCII-Wert des 1. Zeichens) * (2^8)
+ (ASCII-Wert des 2. Zeichens)
   etc.
```

Bei String-SELECT ist ebenfalls nur die Angabe eines maximal vier Zeichen langen Strings hinter CASE zulässig.

Entspricht Expr der CASE-Auswahl, wird die darauffolgende Programmsequenz ausgeführt und daran anschließend direkt zu der auf das zugehörige ENDSELECT folgenden Programmzeile gesprungen. Evtl. weitere CASE-Abfragen derselben Gruppe bleiben dann also unberücksichtigt. Hier ist die SELECT... CASE-Konstruktion mit IF...ELSE IF vergleichbar. Der wesentliche Unterschied ist der, daß der Interpreter die Kontrolle über die Gültigkeit der Auswahl-Vorgaben bei CASE automatisch vornimmt. Die ELSE IF-Konstruktion

```
A%=Random(120)
Print A%,
If A%>5 And A%<44
  Print "innerhalb"          >----------.
Else If A%<6 Or A%>43 And A%<100        |
  Print "außerhalb < 100"    >--------. |
Else If A%>100                        | |
  Print "außerhalb > 100"    >------. | |
Else                                | | |
  Print "100"                >----. | | |
Endif                             | | | |
.<--------------------------------!-!-!-!
```

sieht mit SELECT...CASE folgendermaßen aus:

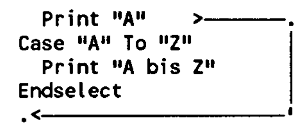
```
A%=Random(120)
Print A%,
Select A%
Case 6 To 43
  Print "innerhalb"          >————————.
Case  To 6,44 To 99                   |
  Print "außerhalb < 100"    >——————. |
Case 101 To                         | |
  Print "außerhalb > 100"    >————. | |
Default                           | | |
  Print "100"                >——. | | |
Endselect                       | | | |
.<——————————————————————————————!-!-!-!
```

Die Verzweigungskriterien hinter CASE sind im Vergleich zu den >-, AND- und OR-Bedingungen der ELSE IF-Struktur durchschaubarer, da sie der Formulierung von Bedingungen im normalen Sprachgebrauch eher entsprechen. Eine weiterer Vorteil gegenüber ELSE IF ist der erhebliche Geschwindigkeitsgewinn:

```
T%=Timer
For I%=1 To 10000       ! 10.000 Durchläufe
  Select I%
  Case 20000            ! 1. Abfrage
  Case 20000            ! 2. Abfrage
  Endselect
Next I%
Print "CASE-Abfrage : ";(Timer-T%)/200;" Sek."
T%=Timer
For I%=1 To 10000       ! 10.000 Durchläufe
  If I%=20000           ! 1. Abfrage
  Else if I%=20000      ! 2. Abfrage
  Endif
Next I%
Print "ELSE IF-Abfrage : ";(Timer-T%)/200;" Sek."
```

An den Sprunglinien im obigen Struktur-Vergleich können Sie erkennen, daß, wenn eine CASE-Bedingung erfüllt ist, anschließend keine der folgenden CASE-Bedingungen durchlaufen wird. So wird im folgenden Beispiel nur die Zeile PRINT "A" ausgeführt, obwohl die folgende CASE-Bedingung "A" To "Z" ebenfalls zutrifft.

```
X$="A"
Select X$
Case "A"
```



Durch die optionale Angabe von TO kann ein ganzer Bereich angegeben werden (z.B. CASE 1 TO 10/CASE "a" TO "z" oder CASE "abc" TO "xyz"), innerhalb dessen Grenzen 'Expr' liegen muß, um die zugehörige Sequenz zu durchlaufen. Wird die erste (kleinere) Bereichsgrenze vor TO oder die zweite (größere) Bereichsgrenze nach TO weggelassen, wird intern automatisch die kleinstmögliche bzw. größtmögliche Grenze angenommen. Durch Verwendung eines Kommas als Trennzeichen können auch mehrere Einzelangaben zusammengefaßt werden (z.B. CASE a,h,j,m oder CASE 1,33,7). Es ist möglich, die CASE-Bedingungsformate in einer CASE-Zeile beliebig zu vermischen (z.B. CASE TO "b","ABC" TO "XYZ",65,66,67,"Ä").

Wurden sämtliche angegebenen CASE-Anweisungen ohne Wahr-Ergebnis passiert, kann am Ende des SELECT-Blocks DEFAULT eingesetzt werden, was dazu führt, daß der zwischen DEFAULT und ENDSELECT liegende Programmteil ausgeführt wird (vergleichbar mit ELSE bei IF-Abfragen).

Wird direkt vor einer CASE-Anweisung am Ende einer Verzweigung die Option CONT verwendet, bewirkt dies, daß die direkt danach stehende CASE-Abfrage übersprungen wird und die dieser CASE-Abfrage unterstellte Sequenz zusätzlich zur schon ausgeführten Verzweigung ebenfalls ausgeführt wird. Nach Erledigung dieser Folgesequenz wird - sofern nicht auch diese mit CONT abgeschlossen wurde - zur ersten Zeile hinter ENDSELECT gesprungen. CONT kann ggf. auch direkt vor DEFAULT eingesetzt werden, was bewirkt, daß - vorausgesetzt, der vor DEFAULT stehende Programmblock wurde ausgeführt - die unter DEFAULT eingefügte Alternativ-Sequenz zusätzlich abgearbeitet wird. Steht CONT nicht direkt vor CASE oder DEFAULT, wird es als CONT zur Programmfortsetzung nach einem STOP-Befehl interpretiert.

Übrigens ist es zwecklos, Programmzeilen zwischen SELECT und dem ersten CASE unterbringen zu wollen, da diese Zeilen vom BASIC nicht registriert werden.

Statt DEFAULT kann auch OTHERWISE { OT } geschrieben werden. Dieser Ausdruck wird vom Interpreter in DEFAULT umgewandelt.

6.3 Bereichsdeklaration

!	Kommentar innerhalb einer Befehlszeile

```
Befehlszeile ! Kommentartext
```

Grundsätzlich hat dieses Ausrufungszeichen die gleiche Aufgabe wie der Befehl REM. Der darauf folgende Text wird als Programm-Kommentar deklariert und von BASIC bei der Interpretation "übersehen". Sie sind dieser Form der Kommentar-Deklaration bei den Beispielprogrammen hier im Buch schon häufiger begegnet.

Der Unterschied zwischen ! und REM besteht darin, daß durch ! der Kommentartext direkt an eine Befehlszeile angehängt werden kann. In DATA-Zeilen ist diese Abgrenzung allerdings nicht möglich, da sie hier als Text-DATA angesehen würde. Beispiel:

```
Input A$      ! String einlesen
Print A$      ! String ausgeben
Edit          ! Programmende
```

Am Anfang einer Zeile verwendet, wird ! in die REM-Abkürzung (') umgewandelt.

DATA { D } — Daten-Speicher deklarieren

```
DATA [Wertedatas [,["] Textdatas ["],...]]
```

Der Anweisung wird ggf. eine Liste durch Kommata getrennter Werte oder Texte übergeben. Sie dient dazu, einem auftretenden READ-Befehl die entsprechende Anzahl von Daten zur Verfügung zu stellen.

Wie bei RESTORE beschrieben, kann ein DATA-Zeiger auf eine bestimmte Marke (Label) gerichtet werden. READ liest dann der Reihe nach die DATAs, die auf die angegebene Marke folgen. Wird kein RESTORE verwendet, werden vom Programmanfang aus nacheinander so viele DATAs eingelesen, wie READ-Anweisungen vorhanden sind. Werden mehr READ-Anweisungen eingesetzt als DATAs bis zum Programmende vorhanden sind, wird eine entsprechende Fehlermeldung ausgegeben.

Eine Besonderheit des GFA-BASICs ist es, daß Text-DATAs nicht zwischen An- und Abführungsstriche gesetzt werden müssen. Es sei denn, daß in Text-DATAs Kommas vorkommen. In diesem Fall würde das Text-Komma als Trenn-Komma zum nächsten String-Teil gewertet werden. Werden die An-/Abführungzeichen vernachlässigt, liest der Interpreter alle vorkommenden Zeichen (auch Leerzeichen), die zwischen den einschließenden Kommas aufgeführt sind.

Numerische READ-Anweisungen sind darauf angewiesen, auch numerische DATAs vorzufinden. Diese können dann allerdings auch in der binären, hexadezimalen oder oktalen Schreibweise angegeben sein. Liest ein Text-READ ein numerisches Zeichen, so wird es einfach als Textzeichen interpretiert. Variablen können in den DATAs nicht übergeben werden. Kommen im Programm keine READ-Anweisungen vor, kann in DATA-Zeilen beliebiger Text stehen. Dieser Text bleibt dann - wie bei REM - vom Programm unberücksichtigt. Schauen Sie sich hierzu bitte auch die Beispiele zu WRITE und TAB() an.

READ { REA } DATA-Werte auslesen

```
READ Var [,Var2,Var$,Var2$,...]
```

Der/den angegebenen Variablen Var werden die jeweils gelesenen DATA-Einträge zugeordnet. Wird kein RESTORE Labelname verwendet, werden der Reihe nach vom Programmanfang aus so viele DATAs eingelesen (falls vorhanden), wie READ-Anweisungen ausgeführt werden. Weiteres siehe unter DATA.

REM { R oder ' oder ! } Kommentar einfügen

```
REM [Kommentar]
```

Möchten Sie Ihr Programm zur besseren Verständlichkeit mit Kommentar versehen, können Sie mit der Anweisung REM an beliebiger Programmstelle eine beliebige Kommentarzeile einfügen. Diese Kommentarzeilen bleiben vom Interpreter im Programmverlauf unberücksichtigt. Als Abkürzung können Sie für REM auch das Hochkomma (Apostroph) verwenden.

Außerdem kann auch ein Ausrufungszeichen (Kommentar-Marker innerhalb einer Befehlszeile) eingesetzt werden. Dieses wird - sofern es sich am Zeilenanfang befindet - in das REM-Zeichen ' umgewandelt.

Es ist empfehlenswert, diese Möglichkeiten zur Kommentierung ausgiebig zu nutzen. Bei größeren Programmen erleichtert es Ihnen das Auffinden von bestimmten Programmteilen. Außerdem machen kleine Bemerkungen die Logik Ihrer Programmführung und evtl. die Zusammenhänge auch für andere überschaubarer.

RESTORE { RES } — DATA-Zeiger setzen

```
RESTORE [Labelname]
```

RESTORE ohne Angabe einer Marke bewirkt, daß der DATA-Zeiger grundsätzlich auf den ersten aller im Programm enthaltenen DATA-Einträge gerichtet wird. Durch RESTORE Labelname kann der DATA-Zeiger auf das erste DATA gerichtet werden, das auf das angegebene Label folgt.

Ab dort werden dann die vorhandenen DATAs eingelesen, bis ein neuer RESTORE-Befehl eingesetzt wird oder das letzte aller DATAs gelesen wurde. Weitere Informationen hierzu finden Sie unter DATA.

6.4 Variablendeklarationen

DEFBIT { DEFBI } — Boole-Variable(n) deklarieren

```
DEFBIT Defstring$
```

DEFxxx-Befehle dienen der globalen Deklaration von Variablentypen. Es können verschiedene Definitions-Strings verwendet werden, die alle Variablen mit der darin angegebenen Namensspezifikation dem entsprechenden Variablentyp zuordnen.

Diese Deklaration wird üblicherweise zu Programmbeginn ausgeführt, da sonst eine Unterscheidung zwischen deklarierten und frei definierten Variablen erschwert wird. Wird jedoch innerhalb des Programms eine neue Deklaration ausgeführt, ist die vorhergehende damit ungültig. Nach einer Deklaration ist es nicht mehr nötig, die dadurch betroffenen Variablennamen mit einem

sogenannten Postfix (Erkennungssymbol, z.B. % für 4-Byte-Integer, ! für Boole-Variablen oder $ für String-Variablen) zu versehen.

Es ist trotzdem jederzeit möglich, einzelne Variablen separat zu definieren, auch wenn sie dieselbe Namensspezifikation wie eine globale Deklaration aufweisen. Dazu ist dem separaten Variablennamen das entsprechende Postfix hinzuzufügen. Damit wird die so unmißverständlich gekennzeichnete Variable von der Deklaration ausgeschlossen. Separat gekennzeichnete Variablen haben generell Vorrang vor den globalen Deklarationen.

Defstring$ ist ein String (Konstante, Variable oder Ausdruck), durch den die Namensspezifikation festgelegt wird. Bei den folgenden Definitionsbeispielen sind die verwendeten Defstring$ beliebig austauschbar.

Definitionen:

DEFBIT "a"

Alle Variablen, deren Name als ersten Buchstaben ein a trägt, sind hiermit - sofern nicht separat definiert (s.o.) - als Boole-Variablen (Var!) deklariert.

DEFBYT "b,c,g-l"

Alle Variablen, deren Name als erstes Zeichen ein b, c, g, h, i, j, k oder l trägt, werden als 1-Byte-Integer (Var|) deklariert.

DEFWRD "word"

Alle Variablen, deren Name mit den Buchstaben word beginnt, werden als 2-Byte-Integer (Var&) deklariert.

DEFINT "i1,i2,i3"

Alle Variablen, deren Name mit den Buchstaben i1, i2 oder i3 beginnt, werden als 4-Byte-Integer (Var%) deklariert.

DEFFLT "a-c,x-z"

Alle Variablen, deren Name als ersten Buchstaben ein a, b, c, x, y oder z trägt, werden als 8-Byte-Fließkommavariablen (Var#) deklariert.

DEFSTR "d-f"

Alle Variablen, deren Name als ersten Buchstaben ein d, e oder f trägt, werden als String-Variablen (Var$) deklariert. Beispiel:

```
Defint "i"          ! Alle mit I beginnenden Variablen
'                   ! gelten ab jetzt als 4-Byte-Integer.
I_string$="XYZ"     ! Beliebige Zuweisung zu einer mit
'                   ! I beginnenden String-Variable.
Print I_string$     ! Es wird XYZ ausgegeben, da trotz
'                   ! der vorangegangenen Definition die
'                   ! direkt angegebene String-Spezifikation
'                   ! $ gültig bleibt.
I_4byte=1548892.88  ! Beliebige Zuweisung zu einer mit I
'                   ! beginnenden Variablen ohne Postfix.
Print I_4byte       ! Es wird 1548892 ausgegeben. Übergebene
'                   ! Nachkommastellen werden integriert. Die
'                   ! 4-Byte-Definition kommt zur Geltung.
```

DEFBYT { DEFB } 1-Byte-Integervariablen deklarieren

```
DEFBYT Defstring$
```

Siehe Erläuterungen zu DEFBIT.

DEFFLT { DEFFL } 8-Byte-Fließkommavariablen deklarieren

```
DEFFLT Defstring$
```

Statt DEFFLT kann im Editor auch DEFSNG oder DEFDBL verwendet werden. Diese Angaben werden vom Editor bei der Syntaxkontrolle in DEFFLT umgewandelt. Sonst siehe DEFBIT.

DEFINT { DEFI } — 4-Byte-Integervariablen deklarieren

```
DEFINT Defstring$
```

Siehe Erläuterungen zu DEFBIT.

DEFSTR { DEFS } — Zeichenkettenvariable(n) deklarieren

```
DEFSTR Defstring$
```

Siehe Erläuterungen zu DEFBIT.

DEFWRD { DEFW } — 2-Byte-Integervariablen deklarieren

```
DEFWRD Defstring$
```

Siehe Erläuterungen zu DEFBIT.

6.5 Unterprogramme

DEFFN — Funktion definieren

```
DEFFN Funkt.name [(Var-Liste)]=Funkt.expr
```

DEFFN ist eine sehr komfortable Möglichkeit, numerische oder alphanumerische Funktionen kompakt zu definieren. "Funkt.name" steht für einen beliebigen Namen, mit dem die Funktion durch FN (bzw. @) angesprochen werden kann. In der Wahl des Funktionsnamens werden Ihnen keinerlei Einschränkungen auferlegt. Sie können außerdem auch BASIC-Befehlsnamen oder

Namen schon existierender Variablen verwenden. Dies ist möglich, da der Funktionsname untrennbar mit der unter ihm definierten Funktion in Bezug steht. Die Verwendung von Feld-Variablennamen ist allerdings nicht zulässig. Das erste Zeichen kann, anders als bei Variablen, auch eine Ziffer sein.

Dem Namen kann optional (wahlfrei) eine Liste von Variablennamen in Klammern angefügt werden (Var-Liste), deren Inhalte dann innerhalb der Funktion verarbeitet werden können. Die einzelnen Variablennamen sind dabei durch Komma voneinander zu trennen. Diese Parameterliste kann ohne weiteres Variablen unterschiedlicher Typen (Real, Integer, String etc.) beinhalten. Innerhalb der Funktion sind dann allerdings nur Operationen erlaubt, die dem verwendeten Funktionstypen entsprechen. D.h., wenn die Funktion als String-Typ deklariert wurde (z.B. DEFFN Funktion$...), können nur String-Operationen ausgeführt werden. Bei numerischen Funktionstypen können folglich nur numerische Operationen eingesetzt werden.

Haben Sie eine Parameterliste vorgesehen, müssen beim Funktionsaufruf dementsprechend viele und dem jeweiligen Variablentyp entsprechende Werte, Strings, andere Variablen oder Ausdrücke übergeben werden. Entsprechen die übergebenen Parametertypen nicht der Liste oder werden zu viele bzw. zuwenig Parameter übergeben, erscheint eine Fehlermeldung.

Die angegebenen Listen-Variablen müssen jedoch nicht zwingend innerhalb der Funktion verwendet werden (Dummy). Andererseits können globale Variablen in der Funktion verwendet werden, die nicht in der Parameterliste stehen. Bei Aufruf der Funktion werden dann die aktuellen Inhalte der darin verwendeten Variablen angenommen und weiterverarbeitet. Existiert eine globale Variable mit gleichem Namen wie der einer Parameter-Aufnahmevariable, so kommt deren Inhalt nicht zur Geltung, da die Listen-Variable innerhalb der Prozedur als lokal verarbeitet wird. D.h. sie hat dann ausschließlich den Wert, der ihr bei Funktionsaufruf über die Parameterliste zugewiesen wurde, und nach Rückkehr wieder den vorherigen globalen Inhalt.

Die Länge einer Funktion wird durch die maximale Eingabezeilenlänge (256 Zeichen) beschränkt. Reicht dieser Platz nicht aus, um eine Funktion komplett zu definieren, können aus einer Funktion heraus andere Funktionen aufgerufen werden. Man kann also Funktionen beliebig miteinander verketten. Eine Funktion kann demnach ohne weiteres ausschließlich aus Funktionsaufrufen und deren Verknüpfungen bestehen.

Funktionen können an jeder beliebigen Stelle des Programms definiert sein. Da bei Programmstart alle DEFFNs initialisiert werden, kann dies also auch am Programmende geschehen, selbst wenn der Funktionsaufruf FN (bzw. @) vor der Funktion auftritt. Zu Beginn des Programmlaufs durchsucht der Interpreter den Programmtext nach evtl. vorhandenen DEFFNs. Deren Inhalt ist ihm daher vor Ausführung der ersten Zeile bereits bekannt.

Vorsicht: Endlosschleifen, worin sich zwei Funktionen gegenseitig aufrufen, können auch durch die Break-Funktion <Control/Shift/Alternate> nicht mehr unterbrochen werden. Rekursive Aufrufe haben denselben Effekt.

FN { @ } — Funktion aufrufen

```
FN Funktionsname [(Parameterliste)]
```

Mit FN bzw. dessen Kürzel @ können selbstdefinierte Funktionen aufgerufen werden. Sollen Parameter an die Funktion übergeben werden, wird dem Aufruf in Klammern eine Parameterliste nachgestellt, innerhalb derer die einzelnen Parameter ggf. durch Kommas voneinander zu trennen sind. Wie bei DEFFN erwähnt, ist darauf zu achten, daß diese Angaben den in der DEFFN-Var-Liste aufgeführten Variablentypen entsprechen.

Das Ergebnis einer Funktion kann - wie bei jeder anderen BASIC-Funktion auch - über einen Ausgabe-Befehl (PRINT, WRITE, TEXT, OUT) ausgegeben, durch eine Zuweisung einer dem Funktionstyp entsprechenden Variablen übergeben und in entsprechende Ausdrücke oder Bedingungsabfragen integriert werden.

FUNCTION..RETURN..ENDFUNC { FU..RET..ENDF }

Funktion

```
FUNCTION Name [(Var1,Var2%,Var3$,...[,VAR Var,...])]
  ... auszuführende Programmteile
  RETURN Back
ENDFUNC
```

FUNCTION kennzeichnet den Anfang einer selbstdefinierten mehrzeiligen Funktion. Es kann eine optionale Liste von lokalen Variablen (wie bei DEFFN und PROCEDURE) angegeben werden, welche ggf. die durch FN (bzw. @) übergebenen Daten aufnehmen. Dabei ist darauf zu achten, daß die Typen der Variablen den übergebenen Parametern entsprechen.

Es ist auch möglich, durch VAR innerhalb dieser Liste Variablen zu definieren, an die dann durch FN eine globale Variable direkt übergeben werden kann (siehe VAR, PROCEDURE). Außerdem ist - wie bei PROCEDURE - die Übergabe von indirekten Pointer-Parametern möglich (siehe auch GOSUB).

"Name" ist ein beliebiger Name, der die Funktion benennt. Innerhalb der FUNCTION kann beliebig viel Programmtext angeordnet werden. Im Gegensatz zu DEFFN-Funktionen ist es möglich, FUNCTION-Funktionen sich selbst aufrufen zu lassen (Rekursion).

Grundsätzlich ist eine FUNCTION mit einer PROCEDURE vergleichbar. Ein Unterschied ist, daß ENDFUNC (wie RETURN bei PROCEDURE) zwar den strukturellen Abschluß einer

FUNCTION-Funktion bildet, jedoch im Gegensatz zu PROCEDURE...RETURN diese Endmarkierung nicht als Rücksprunganweisung interpretiert wird. Aus FUNCTION-Funktionen ist nur ein Rücksprung durch RETURN Back möglich, was nicht mit dem Prozedurende RETURN verwechselt werden darf.

Trifft das Programm innerhalb einer FUNCTION auf die Rücksprunganweisung RETURN Back, wird der hinter RETURN stehende Wert, Ausdruck oder Variableninhalt Back als Funktionsergebnis an das Programm zurückgegeben und zu dem dem Aufruf folgenden Befehl gesprungen. Es können beliebig viele RETURN-Back-Anweisungen innerhalb einer FUNCTION angegeben werden (z.B. innerhalb von IF..ELSEIF..ELSE..ENDIF-Abfragen). Wie auch FN-Aufrufe einzeiliger Funktionen (DEFFN) kann ein Aufruf mehrzeiliger FUNCTIONs ebenfalls beliebig in Ausdrücke und Bedingungen eingebunden bzw. als Variablenzuweisung verwendet werden.

Soll die Funktion alphanumerische (String-)Ergebnisse liefern, ist dem Funktionsnamen ein $ anzuhängen (z.B. FUNCTION Name$). Beispiel:

```
A$="1 ** UPPER/LOWER-Test-String ** 1"
Print "NORMAL: ";A$
Print "UPPER$: ";Upper$(A$)
Print "LOWER$: ";@Lower$(A$)
'
Function Lower$(L.str$)
  ' Untersucht einen vorgegebenen String auf Großbuchstaben.
  ' Gefundene Großbuchstaben werden in die entsprechenden
  ' Kleinbuchstaben umgewandelt.
  ' L.str$ = umzuwandelnder String
  '
  Local L.k%
  For L.k%=1 To Len(L.str$)
    Select Mid$(L.str$,L.k%,1)
    Case "A" To "Z","Ä","Ö","Ü"
      Mid$(L.str$,L.k%,1)=Chr$(Asc(Mid$(L.str$,L.k%,1))+32)
      ' Mid$(L.str$,L.k%,1)=Chr$(Asc(Mid$(L.str$,L.k%,1)) Or 32)
    Endselect
  Next L.k%
  Return L.str$
Endfunc
```

An diesem kleinen Beispiel läßt sich leicht das fast unüberschaubare Einsatzgebiet mehrzeiliger Funktionen erkennen. Ihrer Phantasie sind da keine Grenzen gesetzt. Hätte Frank Ostrowski die RINSTR-Funktion nicht implementiert, wäre sie hiermit spielend zu verwirklichen. Soll hier gleichzeitig der Inhalt von A$ nach Funktionsende gegen den Lower-Case-String ausgetauscht werden, muß man dazu nur den FUNCTION-Kopf folgendermaßen ändern:

```
Function Lower$(Var L.str$)
```

Übrigens soll die REM-Zeile Mid$(...) im Beispiel einen kleinen Trick verdeutlichen. Die beiden MID$()-Zeilen unter "A" To "Z" lassen sich beliebig austauschen. Wissenswert ist, daß der einzige Unterschied zwischen Groß- und Kleinbuchstaben das Bit 5 (2^5=32) des ASCII-Wertes ist. Ist dieses Bit gesetzt, so handelt es sich um Kleinbuchstaben (ASCII: 97 - 122). Ist es gelöscht, so sind es folgerichtig Großbuchstaben (ASCII: 65 - 90).

GOSUB { G oder @ } Verzweigung zu einer PROCEDURE

```
GOSUB Prozedur [(Parameterliste)]
```

Eine GOSUB-Verzweigung springt aus der GOSUB-Zeile direkt zum Kopf der in "Prozedur" bezeichneten PROCEDURE. Ist die Prozedur abgearbeitet, wird vom Prozedurende RETURN aus direkt zu der auf den entsprechenden GOSUB-Aufruf folgenden Programmzeile zurückgesprungen und dort das Programm fortgesetzt.

GFA-BASIC bietet Ihnen wahlweise auch die Möglichkeit, an eine Prozedur eine beliebig lange Parameterliste zu übergeben. Diese Parameter werden dort einer Liste von Variablen zugeordnet, die in Klammern im Prozedurkopf (z.B. PROCEDURE Test(Var,Var$,...)) angegeben sind. Die Anzahl der GOSUB-Pa-

rameter bzw. der PROCEDURE-Aufnahmevariablen ist nur durch die maximale Programmzeilenlänge von 256 Zeichen begrenzt.

Typen und Reihenfolge der Parameter in der Parameterliste können beliebig gemischt werden, solange darauf geachtet wird, daß Anzahl, Typ und Listenplatz der Parameter mit Anzahl, Typ und Listenplatz der in PROCEDURE angegebenen Aufnahmevariablen übereinstimmen.

Durch den Sternchen-Pointer (*Var) können Variablen auch indirekt adressiert werden. Das bedeutet, daß ein in der Parameterliste angegebener Pointer die Adresse seiner Variablen an die Prozedur übergibt und nicht deren Inhalt. Wird nun innerhalb der PROCEDURE der dafür vorgesehenen Aufnahmevariablen ebenfalls durch einen Pointer eine Variablenadresse zugewiesen, so finden Sie nach Abschluß in der übergebenen Variablen den durch den internen Pointer zugewiesenen Wert.

Die im Prozedurkopf aufgeführten Variablen gelten innerhalb der Prozedur als lokale Variablen. D.h., daß die in ihnen enthaltenen Werte oder Strings nur innerhalb dieser Prozedur eingesetzt oder abgefragt werden können. Benennen Sie außerhalb dieser Prozedur globale Variablen mit dem gleichen Namen, so ist BASIC in der Lage, diese von den internen lokalen Variablen zu unterscheiden. Nach Rücksprung aus der Prozedur in das Hauptprogramm werden die globalen Inhalte wieder restauriert.

Zur Bildung des Prozedurnamens sind alle normalen alphanumerischen und numerischen Zeichen (A - Z und 0 - 9) sowie der Tiefstrich (Underscore _) und Punkt erlaubt. Anders als bei Variablennamen kann hier auch eine Ziffer als erstes Zeichen verwendet werden.

Außerdem kann statt eines indirekten Pointer-Parameters (*Var) auch die Variable direkt übergeben werden. Dazu muß an entsprechender Stelle in der PROCEDURE-Aufnahmevariablenliste

eine VAR-Variable gleichen Typs vorgesehen werden (siehe VAR). Ein weiteres GOSUB-Beispiel finden Sie unter PROCEDURE.

GOTO { GOT } — Unbedingter Sprung zu einem Label

```
GOTO Label
```

Dies ist in herkömmlichen BASIC-Dialekten einer der am häufigsten verwendeten Befehle. In GFA-BASIC hält sich dagegen seine Verwendung in Grenzen, da in den überwiegenden Fällen der erzielte Effekt durch GOSUB und FN wesentlich eleganter erreichbar ist.

Er bewirkt nichts weiter, als daß das Programm von der Zeile aus, in der es auf diesen Befehl trifft, zu der Programmzeile verzweigt, die mit dem angegebenen Label-Namen markiert ist. Dieses Label kann sich aus beliebigen Zeichen zusammensetzen. Genau wie bei Prozedurnamen ist es auch möglich, eine Ziffer als erstes Zeichen zu verwenden. Der Label-Name muß mit einem nachgestellten Doppelpunkt abgeschlossen werden.

Dasselbe Label kann übrigens auch als Sprungziel für RESTORE Label verwendet werden. Sprünge in oder aus FOR..NEXT-Schleifen oder PROCEDUREs sind nicht erlaubt. Wird dies versucht, so erscheint eine Fehlermeldung.

LOCAL { LOC } — Lokale Variablen deklarieren

```
LOCAL Loc.var [,Lokale Variablenliste,...]
```

Es können innerhalb einer PROCEDURE - und auch in einer FUNCTION - beliebige Variablen als lokal deklariert werden. Diese können dann ausschließlich in der Prozedur verwendet

werden, in der sie deklariert wurden, bzw. in den von dieser Prozedur aufgerufenen Unter-Prozeduren. Für diese Unter-Prozeduren ist dann die lokale Variable der höheren Ebene gleichbedeutend mit globalen Variablen.

Wird im Hauptprogramm bzw. in Routinen höherer Ebene eine Variable mit gleichem Namen benannt, so ist der Interpreter in der Lage, diese von den lokalen Variablen der aktuellen Routine zu unterscheiden (siehe PROCEDURE).

Werden mehrere Variablen gleichzeitig deklariert, können sie unterschiedlichen Typs sein und sind dann durch Kommata voneinander zu trennen.

ON ... GOSUB — Bedingte Verzweigung zu Prozeduren

```
ON Wert GOSUB Proc1 [,Proc2,Proc3,...]
ON Wert Proc1 [,Proc2,Proc3,...]
```

Mit diesem Befehl lassen sich mehrere Prozeduren zu einer Liste zusammenfassen, die dann je nach angegebenem 'Wert' (numerischer Ausdruck, Variable oder Konstante) aufgerufen werden.

Die Entscheidung darüber, zu welcher Prozedur verzweigt werden soll, wird nur in Einerschritten ab 1 aufwärts getroffen. Eine Verzweigung mit Wert = 0 ist nicht möglich. Die Verzweigung erfolgt also nach folgendem Schema:

```
Wert   >=  1  <  2  ==>  Proc1 aufrufen
Wert   >=  2  <  3  ==>  Proc2 aufrufen
Wert   >=  3  <  4  ==>  Proc3 aufrufen
usw.
```

Ist Wert größer als die Anzahl der angegebenen Prozeduren oder ist Wert gleich null, so wird das Programm ohne Verzweigung in der auf ON GOSUB folgenden Programmzeile fortgesetzt.

Bei sämtlichen ON..GOSUB-Varianten des GFA-BASIC sind nur Prozeduren als Ziel erlaubt, die keine Parameterliste erwarten. Beispiel:

```
On Menu(1)-20 Gosub Top,Clos,Full,Leer,Leer,Leer,Siz,Mov
Procedure Top
  ' Maßnahmen zum Aktualisieren eines Fensters
Return
Procedure Clos
  ' Maßnahmen zum Schließen eines Fensters
Return
Procedure Full
  ' Maßnahmen zum Dimensionieren eines Fensters auf
  ' größtmögliche Ausmaße.
Return
Procedure Siz
  ' Maßnahmen zur beliebigen Dimensionierung eines Fensters
Return
Procedure Mov
  ' Maßnahmen zum Bewegen eines Fensters
Return
Procedure Leer
  ' Dummy-Prozedur ohne Inhalt, die nur dann aufgerufen
  ' wird, sobald ein ON..GOSUB-Wert auftritt, der ohne
  ' Auswirkung bleiben soll.
Return1
```

ON BREAK [CONT] [GOSUB] Break-Funktion behandeln

```
ON BREAK GOSUB Prozedur
ON BREAK
ON BREAK CONT
```

Verzweigt im ersten Fall zu der angegebenen Prozedur, falls nach Ausführung des Befehls im Programmlauf die Break-Funktion <Control><Shift><Alternate> verwendet wird.

Im zweiten Fall wird die normale Break-Funktion aktiviert. D.h., das Programm wird nach <Control><Shift><Alternate> beendet.

Die dritte Syntaxform bewirkt, daß das Programm bis zum nächsten ON BREAK oder ON BREAK GOSUB bzw. bis zum

Programmende nicht mehr durch die Break-Funktion unterbrochen werden kann. Diese Möglichkeit ist sehr vorsichtig zu handhaben, da bei endlos verzweigenden Strukturen ein Programmabbruch dann nur noch über speziell vorgesehene Abbruch-Bedingungen oder durch die gefürchtete Reset-Tastenkombination möglich ist. Beispiel:

```
On break gosub Break        ! Break-Funktion umleiten
Print At(1,1);"1. Stufe"
Print At(1,2);"2. Stufe durch <Control><Shift><Alternate>+<Esc>"
Do                          ! Endlos-Schleife
  If Ex%=1                  ! 2. Stufe schon erreicht?
    If Inkey$=Chr$(13)      ! <Return> gedrückt?
      Cls
      Print At(1,1);"3. Stufe"
      Print At(1,2);"Abbruch durch <Control><Shift><Alternate>"
      On break              ! Break-Funktion wieder einschalten
    Endif
  Endif
Loop
Procedure Break             ! Break-Routine (ist nur
  '                         ! durch Break-Tasten erreichbar)
  If Inkey$=Chr$(27)        ! Vorher <Esc> gedrückt?
    Ex%=1                   ! Flag für Stufe 3 setzen
    Cls
    Print At(1,1);"2. Stufe"
    Print At(1,2);"3. Stufe durch <Return>"
    On break cont           ! Break-Funktion ausschalten
  Endif
Return
```

PROCEDURE { PRO }...RETURN { RET } Prozedur-Titel

Oder:

SUB { SU }...ENDSUB { ENDSU }

Oder:

PROCEDURE { PRO }...ENDPROC { ENDP }

```
PROCEDURE Name [(Variablenliste)]
... auszuführende Programmteile
RETURN
```

Mit PROCEDURE werden in GFA-BASIC Unterprogramme bezeichnet. Diese können beliebigen Programmtext enthalten und werden durch die Rücksprunganweisung RETURN abgeschlossen. Mit RETURN kehrt das Programm zu der aufrufenden Programmzeile zurück und fährt dort mit der darauffolgenden Programmzeile fort.

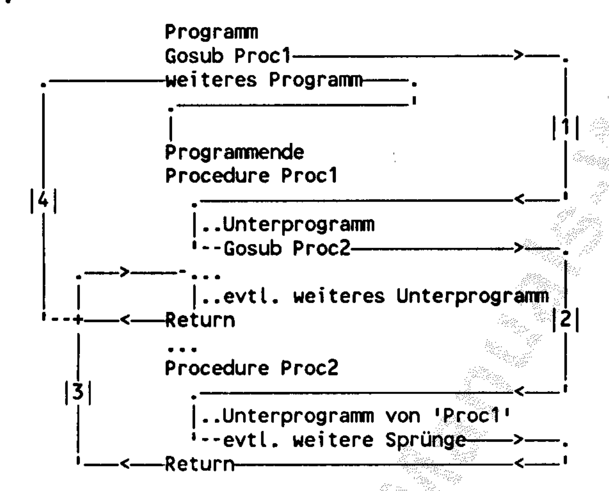
Es ist nicht sehr rationell, bestimmte Programmabläufe, die in immer derselben Form mehrfach im Hauptprogramm erforderlich werden, immer wieder neu zu formulieren. Dieses Verfahren ist also immer dann vorteilhaft, wenn allgemein gehaltene Unterprogramme von mehreren Programmstellen aus aufgerufen werden sollen.

Es kann optional eine Variablenliste lokaler Variablen (siehe LOCAL) angegeben werden, die ggf. die durch GOSUB übergebenen Daten aufnehmen. Die Variablen dieser Liste können unterschiedlichen Typs sein, wobei darauf zu achten ist, daß die Variablentypen den übergebenen Parametertypen entsprechen (bzw. umgekehrt).

Der Name der Prozedur kann aus beliebigen Zeichen (A - Z) und/oder Ziffern (0 - 9), dem Tiefstrich (_) und/oder dem Punkt gebildet werden. Anders als bei Variablennamen kann auch hier das erste Zeichen aus einer Ziffer bestehen.

Es ist auch möglich, Prozeduren sich selbst aufrufen zu lassen (Rekursion) bzw. aus einer Prozedur heraus beliebig weitere Prozeduren aufzurufen.

Struktur:



Es ist auch möglich, durch VAR innerhalb der Variablenliste Variablen zu definieren, an die dann durch GOSUB eine globale Variable direkt übergeben werden kann (siehe VAR). Außerdem kann statt PROCEDURE auch SUB und statt RETURN auch ENDPROC oder ENDSUB verwendet werden. Diese Ausdrücke werden vom Interpreter in PROCEDURE bzw. RETURN umgewandelt.

Beispiele zu diesen beiden Struktur-Befehlen finden Sie in diesem Buch in Hülle und Fülle. Außerdem finden Sie weitere Informationen darüber unter FUNCTION, LOCAL und GOSUB.

VAR — Direkte Variablen-Übergabe

```
PROCEDURE Name([Var1,...] VAR Varname1 [,Varname2$,...])
FUNCTION Name([Var1,...] VAR Varname1 [,Varname2$,...])
```

Als Alternative zu den indirekten Pointer-Variablen können innerhalb der Kopfzeilen von PROCEDUREs und FUNCTIONs in der Liste der lokalen Aufnahme-Variablen mittels VAR auch

Variablen direkt an die Prozedur/Funktion übergeben werden. Die hinter VAR angegebenen Variablennamen stehen dann innerhalb der Prozedur/Funktion stellvertretend für die durch GOSUB oder FN übergebenen Variablen. Dabei ist zu beachten, daß die VAR-Variablen an letzter Stelle der Variablenliste plaziert werden. Es werden dann alle Variablennamen, die hinter VAR stehen, als VAR-Variablen interpretiert.

Die VAR-Variable ist tatsächlich mit der globalen Variable identisch. Sie trägt nur für die Dauer der Prozedur/Funktion den im Kopf angegebenen Namen. Jede Veränderung der Aufnahmevariablen innerhalb der Prozedur wirkt sich also direkt auf die übergebene globale Variable aus.

Inhalte globaler Variablen, die denselben Namen wie die Aufnahmevariable tragen, bleiben erhalten. Der prozedur-/funktionsinterne Name hat also bis zum Rücksprung in das Hauptprogramm lokalen Charakter. Die gleichnamige globale Variable ist jedoch innerhalb der Routine nicht verfügbar. Nach Rücksprung zum Hauptprogramm (bzw. zur nächsthöheren Ebene) wird dann der globale Inhalt wieder restauriert.

Es ist dabei zu beachten, daß die beiden korrespondierenden Variablennamen dem gleichen Typ angehören. Ist das nicht der Fall oder wird eine Konstante oder ein Ausdruck als Parameter übergeben, erscheint die Fehlermeldung "Falsche VAR-Type".

6.6 Assemler-/C-/PRG-Programmaufrufe

C:()	**Maschinenprogramm (C-kompiliert) aufrufen**

```
Var=C:Adressvar ([Parameterliste])
```

Es ist in "Adressvar" die Adresse einer Maschinenroutine und - optional - eine Liste numerischer Parameter in Klammern zu übergeben. Die Übergabe erfolgt nach üblichen C-Konventionen

auf dem Stack. Sollen 32-Bit-Parameter übergeben werden, ist dem jeweiligen Parameter das Kürzel 'L:' voranzustellen. Sonst gilt das Word-Format (16 Bit). Beispiel:

```
VOID C:(L:Para1%,Para2%,W:Para3%,L:Para4%)
```

erzeugt folgenden Stack:

```
run:
move.l    (sp),return  ; = Rücksprungadresse
move.l   4(sp),para1   ; = 1. Parameter Para1% (Long)
move     8(sp),para2   ; = 2. Parameter Para2% (Word)
move    10(sp),para3   ; = 3. Parameter Para3% (Word)
... etc.
... Maschinenprogramm
rts                    ; Nur RTS verwenden !!
```

Sind keine Parameter zu übergeben, ist eine Leerklammer () zu verwenden. Nach Rückkehr kann der Inhalt von D0 (wie in C) entweder direkt ausgegeben (PRINT C:Var()), einer Variablen übergeben (A%=C:Var()) oder in Bedingungsabfragen eingebunden werden (IF C:Var()).

CALL { CAL } Maschinenprogramm (assembliert) aufrufen

```
CALL Adressvar ([Parameterliste])
```

"Adressvar" enthält die Adresse der aufzurufenden Maschinenroutine. "Parameterliste" enthält, durch Kommata getrennt, die evtl. zu übergebenden Parameter. Diese können beliebigen Typs sein, werden jedoch als 32-Bit-Werte interpretiert. Das Maschinenprogramm empfängt auf dem Stack der Reihe nach die Rücksprungadresse, einen 16-Bit-Wert, der die Parameteranzahl enthält, sowie eine 32-Bit-Adresse, ab der die Parameter im 32-Bit-Format aufeinanderfolgend vom BASIC abgelegt wurden. String-Parameter werden hier vom BASIC mit deren Anfangsadresse übergeben. Beispiel:

```
CALL Prog%(17,"Para",A%,B$,*Var%)
```

erzeugt folgenden Stack:

```
run:
move.l     (sp),return   ; = Rücksprungadresse
move      4(sp),anzahl   ; = 5 Parameter
move.l    6(sp),adress   ; = Zeiger auf Parameterblock
move.l   adress,a1       ; = Zeiger nach a1
move    (a1),para1       ; = Parameter 1 in para1
move   4(a1),para2       ; = Parameter 2 in para2
... etc.
... Maschinenprogramm
rts                      ; Nur RTS verwenden!
```

EXEC { EXE } — CLI-Kommando übergeben

```
EXEC Kommando$,In,Out
```

Dieser Befehl erlaubt es, an das sog. CLI (Command Line Interface) ein Kommando zu übergeben, um dadurch (zum Beispiel) ein beliebiges, ablauffähiges (Nicht-GFA-) Programm zu starten.

Das CLI-Kommando wird dazu in Kommando$ übergeben. In und Out stehen für die Kanalnummern (auch 'Filehandle' genannt) der gewünschten Ein- und Ausgabekanäle. Im Normalfall geben Sie hier jeweils -1 an. Dadurch erfolgt die Ein-/Ausgabe von bzw. in ein CLI-Fenster. (Falls Sie GFA-BASIC vom CLI aus gestartet haben, wird das dabei aktive Fenster genommen. Wurde GFA-BASIC von der Workbench aus aufgerufen, so wird ein neues CLI-Fenster geöffnet.) Um die Ein- oder Ausgaben auf ein anderes Gerät bzw. eine andere Datei umzuleiten, müßten Sie im CLI oder in GFA-BASIC mit der DOS-Funktion Open ein entsprechendes Filehandle erzeugen (siehe auch Erklärungen zur DOS-Bibliothek in Kapitel 5) und dieses dann bei In bzw. Out angeben. In den allermeisten Fällen wird das aber nicht erforderlich sein, da die aufgerufenen Programme in der Regel selbst die Einstellung der Ein-/Ausgabekanäle übernehmen.

Theoretisch läßt sich mit EXEC jedes beliebige CLI-Kommando ausführen. In der Praxis am interessantesten dürfte aber der Aufruf von anderen (Anwender-)Programmen sein. Dabei gibt es grundsätzlich zwei Möglichkeiten (die natürlich auch beim Aufruf einfacher CLI-Kommandos wie etwa 'List' bestehen):

- GFA-BASIC wartet, bis das Programm beendet wird.

- Das Programm läuft parallel zu GFA-BASIC. Mit GFA-BASIC selbst kann normal weitergearbeitet werden.

Im ersten Fall enthält Kommando$ nur den Namen des aufzurufenden Programms (ggf. mit Pfadangabe). Beispiel:

```
EXEC "df0:Utilities/Calculator",-1,-1
```

startet das (auf der Workbench-Diskette befindliche) Programm 'Calculator'. GFA-BASIC wartet nun, bis Sie mit ihren Berechnungen fertig sind, d.h. das Fenster von 'Calculator' schließen. Erst danach können Sie mit GFA-BASIC weiterarbeiten.

Im zweiten Fall müssen Sie dem Namen des auszuführenden Programms ein 'Run' voranstellen. Beispiel:

```
EXEC "Run df0:clock",-1,-1
```

startet die (ebenfalls auf der Workbench-Diskette befindliche) Clock. Das Programm läuft nun parallel zu GFA-BASIC.

Die Anzahl der gleichzeitig ablaufenden Programme ist theoretisch nur durch den verfügbaren Speicherplatz begrenzt. Bitte beachten Sie aber, daß jedes zusätzliche Programm auch zusätzliche Rechenzeit kostet. Dadurch kann sich die Ablaufgeschwindigkeit der einzelnen Programme zum Teil beträchtlich verlangsamen.

MONITOR { M } — Maschinen-Programm aufrufen

```
MONITOR [(Parameter)]
```

Parameter enthält optional einen Übergabewert, der in D0 an die aufgerufene Routine geliefert wird.

Der Befehl MONITOR erzeugt ganz banal eine Illegal-Instruction-Exception. Dies ist ein Systemfehler, der immer dann auftritt, wenn in einem Maschinenprogramm versucht wird, einen Befehl auszuführen, den der Amiga nicht kennt. Das passiert meist dann, wenn der PC (ProgramCounter = CPU-Zeiger auf die nächste auszuführende Adresse) aufgrund eines Programmfehlers "in die Wüste" zeigt.

Die Idee hinter MONITOR ist nun, daß erstens in GFA-BASIC eine Illegal-Instruction-Exception unter normalen Umständen unmöglich ist, und zweitens, daß verschiedene Programme diesen Fehler abfangen, um anschließend in einen Debugger, Disassembler, Maschinen-Monitor oder ähnliches zu springen. Ein solches Programm läßt sich also am leichtesten mit MONITOR aufrufen.

Aber Vorsicht: Bei Auslösung einer Illegal-Instruction-Exception springt der Rechner über einen sog. Vektor (Speicheradresse 16) an die Adresse des MONITOR-Programms. Im Normalfall zeigt dieser Vektor auf die Betriebssystemroutine zur Erzeugung der Guru-Meditations. Bevor Sie MONITOR das erste Mal anwenden, müssen Sie also zunächst einmal diesen Vektor (am besten durch das Programm selbst) auf die Adresse des MONITOR-Programms 'verbiegen' lassen! Ansonsten bekommen Sie einen Systemabsturz.

RCALL { RC } Masch.-Programmaufruf m. Registerzugriff

```
RCALL Adresse,Feld%()
```

Ruft ein ab "Adresse" liegendes Maschinenprogramm auf. Dabei kann der Name eines 32-Bit-Integer-Arrays (mind. 16 Elemente) übergeben werden, in dem vor Aufruf beliebige Longwords abgelegt werden können. Die Inhalte der ersten 15 Elemente dieses Feldes (OPTION BASE beachten) werden vor Ausführung der Routine folgendermaßen in die CPU-Register kopiert (bei OPTION BASE 0):

```
Feld%(0) bis Feld%(7)  -->  d0 bis d7
Feld%(8) bis Feld%(14) -->  a0 bis a6
```

Nach Abschluß der Routine (mit RTS) werden die Registerinhalte in der gleichen Reihenfolge wieder in das Feld zurückgeschrieben. Zusätzlich erhält man in Feld%(15) (bei OPTION BASE 0, sonst in Feld%(16)) den aktuellen Userstack-Pointer (SP = a7 - nur Rückgabe!).

RESTORE { RES } DATA-Zeiger setzen

```
RESTORE [Labelname]
```

RESTORE ohne Angabe einer Marke bewirkt, daß der DATA-Zeiger grundsätzlich auf den ersten aller im Programm enthaltenen DATA-Einträge gerichtet wird. Durch RESTORE Labelname kann der DATA-Zeiger auf das erste DATA gerichtet werden, das auf das angegebene Label folgt.

Ab dort werden dann die vorhandenen DATAs eingelesen, bis ein neuer RESTORE-Befehl eingesetzt wird oder das letzte aller DATAs gelesen wurde. Weitere Informationen hierzu finden Sie unter DATA.

7. Textoperationen

7.1 String-Manipulationen

MID$()= — Teil-String zuweisen

```
MID$(Ziel$,Start [,Anz])="Text"
```

Ziel$ gibt eine String-Variable an, in die "Text" ab Start eingesetzt werden soll. Anz gibt optional die Anzahl der Zeichen von Text an, die maximal in Ziel$ eingesetzt werden sollen. Die ursprüngliche Länge von Ziel$ bleibt unverändert. Es werden max. so viele Zeichen von Text eingefügt, wie ab Start in Ziel$ hineinpassen.

Beispiele hierzu finden Sie unter FORM INPUT und FUNCTION.

LSET { LS } — Zeichen(kette) linksbündig einsetzen

```
LSET Ziel$="Text"
```

Text kann eine beliebige Zeichenkette oder String-Variable sein, deren Inhalt linksbündig in Ziel$ eingefügt wird. Dabei bleibt die ursprüngliche Länge von Ziel$ unverändert. Ist Ziel$ kürzer als Text, wird Text bei der Länge von Ziel$ abgeschnitten. Ist Ziel dagegen länger als Text, werden die restlichen Stellen von Ziel$ mit Leerzeichen aufgefüllt.

Ein Beispiel zu LSET finden Sie in Kap. 5.5.1 "Funktionsweise einer Random-Access-Datei".

RSET { RS } Zeichen(kette) rechtsbündig einsetzen

```
RSET Ziel$="Text"
```

RSET ist grundsätzlich mit LSET vergleichbar. Der einzige Unterschied besteht darin, daß "Text" am Ende des Ziel-Strings eingesetzt wird. Beide Befehle finden überwiegend in Random-Acces-Dateien Verwendung, wo es darum geht, daß trotz String-Veränderung die Längen der Eintragszeilen unverändert erhalten bleiben.

7.2 String-Analyse

INSTR() Zeichen(kette) in einem String suchen

```
Var=INSTR(Ziel$,Such$ [,Start] )
Var=INSTR( [Start,] Ziel$,Such$)
```

Liefert einen Wert, der die absolute Position von Such$ in Ziel$ angibt. Ist Such$ in Ziel$ nicht enthalten, wird der Wert 0 zurückgegeben. Sind beide Strings leer, erhält man eine 1. Bei Verwendung der Option Start wird ab der damit angegebenen Zeichenposition gesucht. Bei der Angabe des Such-Strings ist darauf zu achten, daß hier zwischen Groß- und Kleinschreibung unterschieden wird. Beispiel:

```
Ziel$="Kaum zu glauben. Ich trau' meinen Augen nicht."
Such$="au"
@Astring(0,Ziel$,Such$,Ibk$)
For I%=1 To Len(Ibk$) Step 2
  Print Cvi(Mid$(Ibk$,I%,2))
Next I%
Procedure Astring(S.pos%,Z.str$,S.str$,Var Bk$)
  '
  ' Erweiterte INSTR-Routine, die alle (!) Positionen
  ' eines Strings innerhalb eines Ziel-Strings ab einer
  ' evtl. angegebenen Startposition ermittelt und die
```

```
' Positionsliste als MKI$-Daten-String zurückgibt.
'
' S.pos% = String-Position, ab welcher gesucht werden soll.
'          0 wird wie bei INSTR als 1 interpretiert.
' Z.str$ = Zu untersuchender Ziel-String
' S.str$ = Zu suchender String
' Bk$    = VAR-Variable   ist nach Rückkehr
'                         entweder leer (String
'                         nicht gefunden) oder
'                         enthält der Reihe nach
'                         die gefundenen Positionen
'
Local X$,Pos%
Repeat                              ! Such-Schleife >--------.
  Pos%=Instr(S.pos%,Z.str$,S.str$)  ! Position feststellen   |
  If Pos%>0                         ! String enthalten?      |
    X$=X$+Mki$(Pos%)                ! Position eintragen     |
    S.pos%=Pos%+1                   ! Startposition erhöhen  |
  Endif                             !                        |
Until Pos%=0                        ! Kein String mehr?  <---'
Bk$=X$
Return
```

LEFT$() — Linksbündigen Teil-String ermitteln

`Var$=LEFT$(Ziel$ [,Anz])`

Ohne die Option Anz wird das erste Zeichen des Strings Ziel$ geliefert. Bei Verwendung der Option Anz werden ab String-Anfang Anz Zeichen von Ziel$ geliefert. Ist Anz größer als die Länge von Ziel$, so wird Ziel$ komplett als Ergebnis zurückgegeben. Ist Ziel$ ohne Inhalt (""), wird ein Null-String geliefert.

Beispiele zu LEFT$() finden Sie unter PRINT und unter DATA.

LEN() — String-Länge ermitteln

```
Var=LEN(Var$)
```

Liefert die Länge des angegebenen Strings Var$. Beispiele zu LEN() finden Sie unter FORM INPUT, INKEY$(), LINE INPUT, INSTR und an vielen Stellen im Buch verteilt.

MID$() — Beliebigen Teil-String ermitteln

```
Var$=MID$(Ziel$,Start [,Anz])
```

Es wird ein Teil-String von Ziel$ geliefert. Start gibt die Position in Ziel$ an, ab der gelesen werden soll. Wird die Option Anz verwendet, werden ab Start so viel Zeichen geliefert, wie in Anz angegeben sind. Wird Anz nicht verwendet oder ist der in Anz angegebene Wert größer als die Menge der ab Start verbleibenden Zeichen von Ziel$, so wird der gesamte Rest-String ab Start zurückgegeben. Ist Ziel$ ohne Inhalt, wird ein Null-String ("") geliefert.

Anwendungsbeispiele zu MID$() finden Sie unter anderem bei FORM INPUT und RIGHT$().

PRED() — Nächstkleineres ASCII-Zeichen ermitteln

```
Var$=PRED(Expr$)
```

Liefert das nächstkleinere ASCII-Zeichen des ersten Zeichens von Expr$. Ist als Abkürzung für CHR$(ASC(Expr$)-1) einsetzbar. Beispiel:

```
For I%=65 To 76
  Print "ASCII des Zeichens ";Chr$(I%);
  Print " minus 1 = ";Asc(Pred(Chr$(I%)));
  Print " = Zeichen ";Pred(Chr$(I%))
Next I%
```

RIGHT$() — Rechtsbündigen Teil-String ermitteln

`Var$=RIGHT$(Ziel$ [,Anz])`

Ohne die Option Anz wird das letzte Zeichen des Strings Ziel$ geliefert. Bei Verwendung der Option Anz werden vom String-Ende Anz Zeichen von Ziel$ geliefert. Ist Anz größer als die Länge von Ziel$, so wird Ziel$ komplett als Ergebnis zurückgegeben. Ist Ziel$ ohne Inhalt (""), wird ein Null-String zurückgegeben. Beispiel:

```
X$="Dies ist ein DEMO-String zur Vorführung der String-"
X$=X$+"Trennfunktion CUT. Im Rückgabe-String (hier: A$)"
X$=X$+"wird eine Liste von MKI$-Wertepaaren zurückgegeben"
X$=X$+", die der Reihe nach die Positionen und Länge aller "
X$=X$+"extrahierten Strings enthält."
@Cut(X$,", -:.)",20,*A$)        ! Aufruf
For I%=1 To Len(A$) Step 4      ! MKI$-String in 4er-Steps
  '                             ! durchgehen
  Spos%=Cvi(Mid$(A$,I%,2))      ! Teil-String-Position lesen
  Slen%=Cvi(Mid$(A$,I%+2,2))    ! Teil-Stringlänge lesen
  Print Mid$(X$,Spos%,Slen%)    ! Teil-String ausgeben
Next I%
'
Procedure Cut(C.str$,C.sgn$,C.brt%,C.vec%)
  ' Teilt einen vorgegebenen String in Teile mit einer
  ' vorgegebenen max. Länge, wobei eine Liste von
  ' Trennzeichen angegeben werden kann, die dann Priorität
  ' vor der Länge haben. Der Ausgangs-String bleibt unverändert.
  '
  ' C.str$  =  Zu teilender Vorgabe-String
  ' C.sgn$  =  Beliebige Liste von Trennzeichen. Die
  '            Teil-Strings enden jeweils mit dem ersten
  '            Zeichen hinter dem gefundenen Trennzeichen.
  '            Ist C.sgn$ leer (""), gilt für die
  '            Trennung ausschließlich C.brt%.
  ' C.brt%  =  Max. neue Zeilenbreite
  ' C.vec%  =  Pointer auf eine Rückgabe-String-Variable, die
  '            nach Abschluß eine Liste von MKI$-Wertepaaren
```

```
  '             enthält, die der Reihe nach Position und Länge
  '             der einzelnen Teil-Strings angeben.
  '
  Local C.dum$,Cd.vec$,C.pos%,C.a$,C.j%
  C.pos%=1                      ! Positions-Puffer +1
  Do
    C.a$=Left$(C.str$,Min(C.brt%,Len(C.str$))) ! Teil-String
    '                           ! auf Länge trimmen
    If Len(C.sgn$)>0            ! Trennzeichen vorhanden?
      For C.j%=Len(C.a$) Downto 1 ! Alle Zeichen durchgehen
        Exit if Instr(C.sgn$,Mid$(C.a$,C.j%,1)) ! Abbruch,
        '                       ! wenn Trennzeichen gefunden
      Next C.j%                 ! Nächstes Zeichen
    Endif
    If C.j%=0                   ! Kein Trennzeichen?
      C.j%=C.brt%               ! Längenvorgabe hat Priorität
    Endif
    C.dum$=Left$(C.a$,Min(C.brt%,C.j%)) ! Teil-String "cutten"
    C.str$=Right$(C.str$,Len(C.str$)-Min((C.brt%),Len(C.dum$)))
    '                           ! Ausgangs-String kürzen
    Cd.vec$=Cd.vec$+Mki$(C.pos%)+Mki$(Len(C.dum$))
    '                           ! MKI$-String bilden
    Add C.pos%,Len(C.dum$)      ! Neue Startposition
    Exit if Len(C.str$)<C.brt% ! Exit, wenn Rest < max. Breite
  Loop
  Cd.vec$=Cd.vec$+Mki$(C.pos%)+Mki$(Len(C.str$)) ! MKI$-Rest
  *C.vec%=Cd.vec$               ! MKI$-String zurückgeben
Return
```

RINSTR() Zeichen(kette) in einem String rückwärts suchen

```
Var=RINSTR(Ziel$,Such$ [,Start] )
Var=RINSTR( [Start,] Ziel$,Such$)
```

Liefert einen Wert, der die erste gefundene Position von 'Such$' in Ziel$ angibt. Bei der Durchsuchung des Ziel-Strings wird im Gegensatz zu INSTR am String-Ende begonnen. Weiteres siehe INSTR.

SUCC()	**Nächstgrößeres ASCII-Zeichen ermitteln**

```
Var$=SUCC(Expr$)
```

Liefert das nächstgrößere ASCII-Zeichen des ersten Zeichens von Expr$. Ist als Abkürzung für CHR$(ASC(Expr$)+1) einsetzbar. Beispiel:

```
For I%=65 To 78
  Print "ASCII des Zeichens ";Chr$(I%);
  Print " plus 1  = ";Asc(Succ(Chr$(I%)));
  Print " = Zeichen ";Succ(Chr$(I%))
Next I%
```

7.3 String-Formatierung

SPACE$()	**Leerzeichen-String bilden**

```
Var$=SPACE$(Anz)
```

Es wird ein String aus Anz Leerzeichen gebildet und zurückgegeben.

STRING$()	**Mehrfach-Zeichenkette bilden**

```
Var$=STRING$(Anz,"Text")
Var$=STRING$(Anz,Ascii)
```

Es wird ein String gebildet, der sich daraus ergibt, daß Text so oft verkettet wird, wie in Anz angegeben. Text kann auch als Variable oder String-Konstrukt angegeben werden. Soll ein einzelnes Zeichen multipliziert werden, kann statt Text auch der ASCII-Wert des Zeichens verwendet werden. Die entstehende

Zeichenkette darf nicht mehr als 32767 Zeichen enthalten (maximale Größe einer String-Variablen).

Beispiele finden Sie unter anderem unter PRINT.

TRIM$() Space-Zeichen eliminieren

```
Var$=TRIM$(Ziel$)
```

Eliminiert alle Spaces (Leerzeichen), die am String-Anfang oder am String-Ende von Ziel$ stehen, und liefert den verbleibenden String-Teil. Besteht der übergebene String Ziel$ nur aus Leerzeichen oder ist er ohne Inhalt, erhält man einen Leer-String ("") zurück. Beispiel:

```
For I%=1 To 4
  Read A$
  Print "Vorher : ->";A$;"<-",
  Print "Nachher: ->";Trim$(A$);"<-"
Next I%
Data   String 1  ,  String 2,String   3,String 4   ,
```

UPPER$() Buchstabenumwandlung klein => groß

```
Var$=UPPER$(Ziel$)
```

Trifft UPPER$ beim Lesen von Ziel$ auf einen kleingeschriebenen Buchstaben (ASCII-Werte 97 bis 129), so wird dieser in den entsprechenden Großbuchstaben (ASCII-Werte 65 bis 90) umgewandelt. Kleingeschriebene Umlaute (ä, ö, ü) werden ebenfalls in Uppercase (Ä, Ö, Ü) gewandelt. Alle anderen Zeichen bleiben unverändert.

8. Arithmetik-Befehle

8.1 Operatoren

Arithmetische Operatoren:

^	Potenzieren
-	Negatives Vorzeichen
* /	Multiplikation und Division
DIV	Ganzzahldivision (Entfernen von Dezimalstellen)
MOD	Modulo-Berechnung (Rest der Ganzzahldivision)
+ -	Addition und Subtraktion

Vergleichsoperatoren:

=	ungleich < > oder > < gleich
>	größer als > = oder = > größer oder gleich
<	kleiner als < = oder = < kleiner oder gleich
= =	ungefähr gleich (28 Bit-Vergleich)

Logische Operatoren:

AND	Konjunktion: Das Ergebnis von AND ist wahr, wenn beide Argumente wahr sind.
OR	Disjunktion: Das Ergebnis von OR ist wahr, wenn eines der Argumente wahr ist.
XOR	Exklusives Oder: Das Ergebnis von XOR ist falsch, wenn die Argumente beide wahr oder beide falsch sind.
NOT	Negation: Vertauscht Wahrheitswerte ins Gegenteil.
IMP	Implikation: Die Folgerung IMP ist nur dann falsch, wenn aus etwas Wahrem etwas Falsches folgt.
EQV	Äquivalenz: Umkehrung zu XOR. Das Ergebnis ist falsch, wenn sich die beiden Argumente unterscheiden.

Prioritäten:

()	Klammern (höchste Priorität)
^	Potenzierung
-	Negatives Vorzeichen
* /	Multiplikation und Division
DIV MOD	Ganzzahldivision und Modulo-Berechnung
+ -	Addition und Subtraktion
= > < = =	
<> >= <=	Vergleichsoperatoren
NOT AND OR,	
XOR IMP EQV	Logische Operatoren (niedrigste Priorität)

8.2 Mathematische Operationen

ADD { AD } — Additionsbefehl

```
ADD Var,Wert
```

Addiert Wert zum Inhalt der numerischen Variablen Var und legt anschließend das Ergebnis in Var ab.

Wird in Var eine Integer-Variable angegeben, so wird ein evtl. in Wert angegebener Realwert als Integerwert behandelt.

DEC — Dekrementierung

```
DEC Var
```

Vermindert den Wert der numerischen Variablen Var um den Wert 1.

DIV — Divisionsbefehl { DI }

```
DIV Var,Wert
```

Dividiert den Inhalt der numerischen Variablen Var durch Wert und legt anschließend das Ergebnis in Var ab. Beachten Sie bitte die Anmerkung zum ADD-Befehl.

INC { IN } — Inkrementierung

```
INC Var
```

Erhöht den Wert der numerischen Variable Var um den Wert 1.

MUL { MU } — Multiplikationsbefehl

```
MUL Var,Wert
```

Multipliziert den Inhalt der numerischen Variablen Var mit Wert und legt anschließend das Ergebnis in Var ab. Beachten Sie bitte die Anmerkung zum ADD-Befehl.

SUB { SU } — Subtraktionsbefehl { SU }

```
SUB Var,Wert
```

Subtrahiert Wert vom Inhalt der numerischen Variablen Var und legt anschließend das Ergebnis in Var ab. Beachten Sie bitte die Anmerkung zum ADD-Befehl.

ADD() Integer-Additionsfunktion

```
Var=ADD(Wert1,Wert2)
```

Addiert Wert1 zu Wert2 und liefert dann das entsprechende Integer-Ergebnis.

Anmerkung: Im Gegensatz zu den ADD-, SUB-, MUL-, DIV-Befehlen können diese Funktionen - wie jede andere Funktion auch - in die Ausgabe, in Zuweisungen, Ausdrücke, Bedingungsabfragen etc. eingebunden werden. Sie stehen stellvertretend für das Ergebnis, das sie liefern sollen. Außerdem können hiermit generell nur integrierte Werte berechnet werden. Von ggf. aus der Operation entstehenden Realwerten wird nur der Vorkomma-Anteil geliefert. Eine Rundung findet nicht statt.

Die Werte können als Konstante, Variable, Ausdruck oder Funktion angegeben werden. Sie werden durch die Operation nicht verändert. Gibt man die Werte als Realwerte an, werden sie vor Ausführung der Operation auf ihren Integeranteil reduziert. Beispiel:

```
Print Mod(Add(Div(100,33),Mul(345,Sub(12,5.6))),38)
```

entspricht

```
Print Int(100/33)+Int(345*Int(12-Int(5.6))) Mod 38
```

DIV() Integer-Divisionsfunktion

```
Var=DIV(Wert1,Wert2)
```

Dividiert Wert1 durch Wert2 und liefert das entsprechende Integer-Ergebnis. Beachten Sie bitte die Anmerkung zur ADD()-Funktion.

MOD() Integer-Modulo-Funktion

```
Var=MOD(Wert1,Wert2)
```

Berechnet den ganzzahligen Rest der Integer-Divison Wert1 durch Wert2 (Modulo-Berechnung) und liefert das entsprechende Integer-Ergebnis. Beachten Sie bitte die Anmerkung zur ADD()-Funktion.

MUL() Integer-Multiplikationsfunktion

```
Var=MUL(Wert1,Wert2)
```

Multipliziert Wert1 mit Wert2 und liefert das entsprechende Integer-Ergebnis. Beachten Sie bitte die Anmerkung zur ADD()-Funktion.

SUB() Integer-Subtraktionsfunktion

```
Var=SUB(Wert1,Wert2)
```

Subtrahiert Wert2 von Wert1 und liefert das entsprechende Integer-Ergebnis. Beachten Sie bitte die Anmerkung zur ADD()-Funktion.

8.3 Numerische Funktionen

ABS() **Betrags-Funktion**

```
Var=ABS(Arg)
```

ABS gibt das Argument Arg vorzeichenlos als positiven (absoluten) Wert zurück. Dieser Wert ist immer gleich oder größer null.

EVEN() **Zahl auf "gerade" testen**

```
Var=EVEN(Arg)
```

Liefert -1, wenn Arg gerade ist. Sonst wird 0 geliefert.

EXP() **Exponential-Funktion**

```
Var=EXP(Arg)
```

Berechnet das Ergebnis des Exponenten Arg zur Basis e (Eulersche Zahl). Gleichbedeutend mit: 2.718281828462 ^ Arg. Arg muß auf jeden Fall größer null sein, da sonst eine Fehlermeldung produziert wird. EXP ist die Umkehrfunktion zu LOG.

FIX() **Ganzzahl-Funktion**

```
Var=FIX(Arg)
```

Übergibt den ganzzahligen Anteil (Integer) der Real-Zahl Arg. Die Funktion rundet Zahlen weder auf noch ab, sondern ent-

fernt nur die Dezimalstellen. Im Minusbereich wird dadurch der Minuswert kleiner. FIX(-12.33) ergibt -12. FIX(33.17) ergibt 33. FIX ist identisch mit TRUNC.

FRAC() — Dezimalstellen-Funktion

```
Var=FRAC(Arg)
```

Liefert den Dezimalanteil (Nachkommastellen) von Arg, falls Arg eine reelle Zahl ist, bzw. den Wert 0, falls Arg integer ist.

INT() — Ganzzahl-Funktion

```
Var=INT(Arg)
```

Wandelt die Zahl Arg in eine Integer-Zahl. Ist Arg ein Realwert, so wird die nächstkleinere Ganzzahl zurückgegeben. Im Gegensatz zu FIX() und TRUNC() wird dadurch im Minusbereich der Minuswert größer.

```
INT(-12.33) ergibt -13          INT(33.17) ergibt 33.
```

LOG() — Logarithmus-Funktion

```
Var=LOG[10](Arg)
```

LOG() gibt den natürlichen (Neperschen) Logarithmus des in Klammern angegebenen numerischen Ausdrucks Arg zur Basis e zurück.

```
Basis e => 2.718281828 => Eulersche Zahl
```

Das Ergebnis von 2.718281828462^LOG(Arg) ergibt Arg. LOG10() liefert dagegen den dekadischen (Briggsschen) Logarithmus des in Klammern übergebenen Arguments Arg zur Basis 10.

Das Ergebnis von 10^LOG10(Arg) ergibt Arg. Arg muß auf jeden Fall größer null sein, da sonst eine Fehlermeldung produziert wird. LOG() ist die Umkehrfunktion zu EXP().

ODD() — Zahl auf "ungerade" testen

```
Var=ODD(Arg)
```

Liefert -1, wenn Arg ungerade ist. Sonst wird 0 geliefert.

PRED() — Nächstkleinere Ganzzahl ermitteln

```
Var=PRED(Arg)
```

Liefert die nächstkleinere Integerzahl vor Arg. Bei Arg als Realzahl werden vor Ausführung der Operation die Nachkommastellen integriert (siehe INT).

ROUND() — Rundungsfunktion

```
Var=ROUND(Arg [,Stelle] )
```

Rundet Arg mathematisch exakt auf eine ganze Zahl. Wird die Option Stelle verwendet, wird auf die angegebene Nachkommastelle gerundet. Ist Stelle negativ, wird auf die entsprechende Stelle vor dem Komma gerundet.

SGN() — Vorzeichen ermitteln

```
Var=SGN(Arg)
```

Liefert das Vorzeichen (engl.: SiGN) von Arg.

```
1 wenn Arg > 0/-1 wenn Arg < 0/0 wenn Arg = 0
```

SQR() — Wurzel-Funktion

```
Var=SQR(Arg)
```

Es wird die 2. Wurzel (Quadratwurzel engl.: SQuare Radical) von Arg geliefert. Wird eine höhere Wurzel gebraucht, so ist diese über den Umweg der Potenzierung mit gebrochenem Exponenten zu berechnen:

```
3. Wurzel = Arg^(1/3)
4. Wurzel = Arg^(1/4)
```

etc.

Arg muß auf jeden Fall gleich oder größer null sein, da sonst eine Fehlermeldung produziert wird.

SUCC() — Nächstgrößere Ganzzahl ermitteln

```
Var=SUCC(Arg)
```

Liefert die nächstgrößere Integerzahl nach Arg. Bei Arg als Realzahl werden die Nachkommastellen integriert (siehe INT).

TRUNC() **Ganzzahl-Funktion**

```
Var=TRUNC(Arg)
```

TRUNC ist identisch mit FIX. Weiteres siehe dort.

8.4 Trigonometrische Funktionen

ACOS() **Arcuscosinus-Funktion**

```
Var=ACOS(Arg)
```

Berechnet den Arcuscosinus von Arg. Weiters siehe unter ATN().

ASIN() **Arcussinus-Funktion**

```
Var=ASIN(Arg)
```

Berechnet den Arcussinus von Arg. Weiteres siehe unter ATN().

ATN() **Arcustangens-Funktion**

```
Var=ATN(Arg)
```

Das Argument Arg ist ein Tangenswert, aus dem der ihm entsprechende Winkel in Radiant berechnet wird. Es wird ein Wert zwischen -PI/2 und +PI/2 geliefert. Benötigt man die Winkelangabe in Grad, so muß das Ergebnis mit 180/PI multipliziert bzw. durch DEG umgewandelt werden.

COS() Cosinus-Funktion

`Var=COS(Arg)`

Berechnet den Cosinus von Arg. Arg ist ein Winkel in Radiant. Soll der Winkel in Grad eingegeben werden, muß dieser vorher mit PI/180 multipliziert bzw. durch RAD gewandelt werden. Beispiel:

```
For I=0 To 90 Step 15
  Print "Sinus   ";I;" GRAD : ";Sin(I*Pi/180)
  Print "Cosinus ";I;" GRAD : ";Cos(I*Pi/180)
  Print "Tangens ",I;" GRAD : ";Tan(I*Pi/180)
Next I
For I=-Pi+0.0000001 To Pi Step Pi/90
  Print "Sinus   ";I;"  RAD : ";Sin(I)
  Print "Cosinus ";I;"  RAD : ";Cos(I)
  Print "Tangens ";I;"  RAD : ";Tan(I)
Next I
```

Ein weiteres Beispiel zu COS() und SIN() finden Sie unter PI.

COSQ() Interpolierte Cosinus-Funktion mit Grad-Angabe

`Var=COSQ(Grad)`

Ermittelt den 16tel-Grad-interpolierten Cosinus des in Grad angegebenen Winkels Grad (ca. zehnmal schneller als COS()).

DEG() Umwandlung in Grad

`Var=DEG(Radiant)`

Rechnet die Radiant-Winkelangabe in Gradmaß (DEGree) um. Entspricht dem Ergebnis von ARG*180/PI.

PI Kreiszahl

PI

Reservierte Variable. Steht dort, wo sie eingesetzt wird, für die konstante Kreiszahl PI (3.1415926536).

Grafik: Cbox

Beispiel:

```
Yt%=2                              ! Y-Auflösungsteiler
Deffill ,2,4                       ! DEFFILL grau
For I%=0 To 360 Step 12            !——.
  @Cbox(2,320,200/Yt%,250,I%)         !- PBOX-Schleife
Next I%                            !——'
Cls                                ! Bildschirm klar
For I%=0 To 720 Step 12            !——.
  Add J%,3           ! Radius +3      !- BOX-Schleife
  @Cbox(1,320,200/Yt%,10+J%,I%)    !——'
Next I%
'
Procedure Cbox(Mod%,Xp%,Yp%,Rd%,Wi%)
  ' Produziert ein Quadrat, das in einem beliebigen Winkel
  ' und mit beliebiger Größe dargestellt werden kann.
  ' Mod%    = Darstellungsmodus
  '             1 = BOX
  '             2 = PBOX
  ' Xp%/Yp% = Koordinaten des Mittelpunktes (Drehpunktes)
  ' Rd%     = Umkreisradius
  ' Wi%     = Neigungswinkel
  '
  Local J%,I%,Yt%,Dg
  Yt%=2                          ! Y-Auflösungsteiler
  Erase Px%()                    ! POLY-X-Feld löschen
  Erase Py%()                    ! POLY-Y-Feld löschen
  Dim Px%(4),Py%(4)              ! POLY-Felder dimensionieren
  For I%=-Wi%+45 To -Wi%+360+45 Step 90 ! Einmal rundum
    Dg=I%*Pi/180                 ! In Radiant umrechnen
    Px%(J%)=Xp%+(Sin(Dg)*Rd%*Sqr(2)/2+0.5)     !- Koordinaten..
    Py%(J%)=Yp%+(Cos(Dg)*Rd%/Yt%*Sqr(2)/2+0.5) !- ..berechnen
    '
    Inc J%                       ! Ecken-Zähler +1
  Next I%                        ! Nächste Ecke
  If Mod%=1                      ! BOX-Darstellung?
    Polyline 5,Px%(),Py%()       ! Dann Polyline
  Endif
  If Mod%=2                      ! PBOX-Darstellung?
```

```
      Polyfill 5,Px%(),Py%()    ! Dann Polyfill
    Endif
  Return
```

RAD() Umwandlung in Radiant (Bogenmaß)

```
Var=RAD(Grad)
```

Rechnet die Grad-Winkelangabe Grad in Bogenmaß (RADiant) um.

SIN() Sinus-Funktion

```
Var=SIN(Arg)
```

Berechnet den Sinus von Arg. Weiteres siehe unter COS().

SINQ() Interpolierte Sinus-Funktion mit Grad-Angabe

```
Var=SINQ(Grad)
```

Ermittelt den 16tel-Grad-interpolierten Sinus des in Grad angegebenen Winkels Grad (ca. zehnmal schneller als SIN()).

TAN() Tangens-Funktion

```
Var=TAN(Arg)
```

Berechnet den Tangens von Arg. Weiteres siehe unter COS().

8.5 Vergleichsoperationen

MAX() Größten Wert ermitteln/größten String ermitteln

```
Var=MAX(Expr1,Expr2 [,Expr3,...])
Var$=MAX(Expr1$,Expr2$ [,Expr3$,...])
```

Gibt den größten Wert einer Werteliste bzw. den größten String einer String-Liste zurück. Expr kann ein beliebiger numerischer- oder Textausdruck, Wert, String bzw. Variable sein. Alle Ausdrücke müssen demselben Typ angehören.

Bei String-Vergleichen wird der größte String ermittelt, indem der Reihe nach alle Einzelzeichen der zu vergleichenden Strings überprüft werden. Haben beide Zeichen denselben ASCII-Wert, werden solange die nächsten beiden Zeichen geprüft, bis sie sich unterscheiden oder einer der beiden Strings keine Zeichen mehr enthält. Im ersten Fall ist der Ausdruck größer, dessen zuletzt geprüftes Zeichen den größeren ASCII- Wert besitzt. Im zweiten Falle ist es der String mit der größeren Länge.

MIN() Kleinsten Wert/String ermitteln

```
Var=MIN(Expr1,Expr2 [,Expr3,...])
Var$=MIN(Expr1$,Expr2$ [,Expr3$,...])
```

Gibt den kleinsten Wert einer Werteliste bzw. den kleinsten String einer String-Liste zurück. Siehe auch Erläuterungen zu MAX().

8.6 Bit-Operationen

AND() — Konjunktions-Funktion

```
Var=AND(Wert1,Wert2)
```

Verknüpft logisch die beiden angegebenen Werte im AND-Modus und liefert das Integer-Ergebnis (siehe Kapitel 4 "Basis-BASIC").

Die Booleschen Funktionen stehen jeweils als Ersatz für die entsprechende logische Verknüpfungsoperation:

```
AND(x,y)      entspricht      (x AND y)
 OR(x,y)      entspricht      (x OR y)
XOR(x,y)      entspricht      (x XOR y)
EQV(x,y)      entspricht      (x EQV y)
IMP(x,y)      entspricht      (x IMP y)
```

Beispiel:

```
Print &X11101101 And &X1111'''''And(%11101101,%1111)
Print &HED Or &HF'''''Or($ED,$F)
Print 237 Xor 15'''''Xor(237,15)
```

Durch die Funktionsform werden diese Operationen für den Programmierer übersichtlicher und für das BASIC schneller "erfaßbar", wodurch sich Geschwindigkeitsvorteile von bis zu 10 Prozent ergeben.

In der Beschreibung zu IF..ENDIF habe ich den Vorgang der Bedingungsstellung durch AND und OR zu erklären versucht. Eine häufige Bedingungsform ist z.B.:

```
IF ((v1%=w1 OR v2%<>w1) AND v3%>(ex1%+w2)) OR v4%<w2
```

Durch die Verknüpfungsfunktionen kann diese Bedingung folgendermaßen umgestaltet werden:

```
IF OR(AND(OR(v1%=w1,v2%<>w1),v3%>(ex1%+w2)),v4%<w2)
```

Durch die klare Klammersetzung kann ein Ausdruck dieser Art wesentlich leichter analysiert werden. Ein kleiner Nachteil ist, daß OR- oder AND-Ketten etwas unübersichtlicher werden.

Aus:

```
IF v1%>w1 OR v2%<>w1 OR v3%<>v1% OR v4%=w2
```

wird:

```
IF OR(OR(OR(v1%>w1,v2%<>w1),v3%<>v1%),v4%=w2)
```

Aber das ist wohl Geschmacksache. Man wird ja durch nichts daran gehindert, in solchen Fällen die erste Variante zu verwenden.

BCHG() Einzel-Bit wechseln (Xor-en)

```
Var=BCHG(Wert,Bit)
```

Wechselt das angegebene Bit von Wert (Wert XOR 2^ Bit). War das Bit vorher 1, dann ist es anschließend 0 und umgekehrt.

Beachten Sie bei den Einzel-Bit-Funktionen, daß die Bit-Zählung von rechts mit der Bit-Nummer 0 beginnt. Bit kann beliebig angegeben werden, wobei größere Werte als 31 (max. Breite=32 Bit) nicht mehr korrekt verarbeitet werden. Größere Angaben für Wert als 2^32-1 (&HFFFFFFFF = max. Longwert) werden nicht verarbeitet.

BCLR() — Einzel-Bit löschen

```
Var=BCLR(Wert,Bit)
```

Löscht das angegebene Bit von Wert (Wert AND NOT 2^Bit). War das Bit vorher 1, ist es anschließend 0. 0 bleibt 0. Weiteres siehe unter BCHG().

BSET() — Einzel-Bit setzen

```
Var=BSET(Wert,Bit)
```

Setzt das angegebene Bit von Wert (Wert OR 2^Bit). War das Bit vorher 0, ist es anschließend 1. War es 1, bleibt es 1. Weiteres siehe unter BCHG().

BTST() — Einzel-Bit auf an/aus testen

```
Var=BTST(Wert,Bit)
```

Testet das angegebene Bit von Wert (-SGN(Wert AND 2^Bit)). Ist das Bit gesetzt, wird eine -1 (TRUE) geliefert, sonst 0 (FALSE). Weiteres siehe unter BCHG().

BYTE() — Vorzeichenloses LO-Byte eines Wertes liefern

```
Var=BYTE(Wert)
```

Liefert vorzeichenlos die untersten 8 Bit von Wert.

CARD() — Vorzeichenloses LO-Word eines Wertes liefern

```
Var=CARD(Wert)
```

Liefert vorzeichenlos die untersten 16 Bit von Wert.

EQV() — Äquivalenz-Funktion

```
Var=EQV(Wert1,Wert2)
```

Verknüpft logisch die beiden angegebenen Werte im EQV-Modus und liefert das Integer-Ergebnis. Weiteres siehe unter AND().

IMP() — Implikations-Funktion

```
Var=IMP(Wert1,Wert2)
```

Verknüpft logisch die beiden angegebenen Werte im IMP-Modus und liefert das Integer-Ergebnis. Weiteres siehe unter AND().

OR() — Disjunktions-Funktion

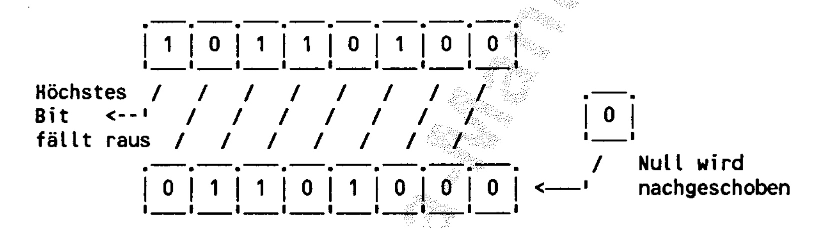
```
Var=OR(Wert1,Wert2)
```

Verknüpft logisch die beiden angegebenen Werte im OR-Modus und liefert das Integer-Ergebnis. Weiteres siehe unter AND().

SHL() Bits links verschieben

```
Var=SHL(Wert,Bits) Long-Shift-Left
Var=SHL&(Wert,Bits) Word-Shift-Left
Var=SHL¦(Wert,Bits) Byte-Shift-Left
```

Verschiebt den Inhalt von 'Wert' um die Anzahl 'Bits' nach links. Je verschobenem Bit wird 'Wert' dabei mit 2 multipliziert (Integer-Multiplikation: Var=Wert*2^Bits). Das rechts frei werdende Bit wird mit 0 gefüllt.



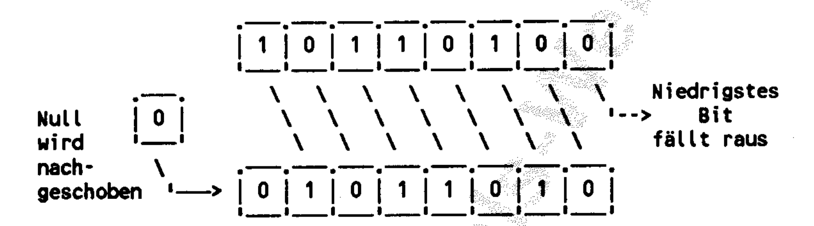
Bei Angabe von & hinter SHL werden nur die ersten 16 Bit (LO-Word) von Wert verschoben, bei | nur die ersten 8 Bit (LO-Byte). Ist Wert bei SHL|() größer als 8 Bit, so werden als Ergebnis trotzdem nur die untersten 8 Bit der Operation geliefert. Dasselbe gilt für SHL&() (16 Bit), wobei dann jedoch das Bit 15 des Ergebnisses in die Bits 16-31 kopiert wird (vorzeichenbehaftet).

SHR() Bits logisch rechts verschieben

```
Var=SHR(Wert,Bits)Long-Shift-Right
Var=SHR&(Wert,Bits) Word-Shift-Right
Var=SHR¦(Wert,Bits) Byte-Shift-Right
```

Verschiebt den Inhalt von 'Wert' um die Anzahl 'Bits' nach rechts. Je verschobenem Bit wird 'Wert' dabei durch 2 dividiert (Integer-Division: Var=Wert DIV 2^Bits). Das links frei wer-

dende Bit wird mit 0 gefüllt. Bei Komplementwerten bleibt also das Vorzeichen nicht erhalten.

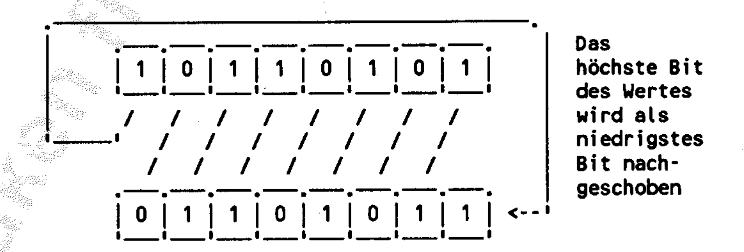


Bei Angabe von & hinter SHR werden nur die ersten 16 Bit (LO-Word) von Wert verschoben, bei | nur die ersten 8 Bit (LO-Byte). Ist Wert bei SHR|() größer als 8 Bit, so werden als Ergebnis trotzdem nur die untersten 8 Bit der Operation geliefert. Dasselbe gilt für SHR&() (16 Bit), wobei dann jedoch das Bit 15 des Ergebnisses in die Bits 16-31 kopiert wird (vorzeichenbehaftet).

ROL() — Bits links rotieren

```
Var=ROL(Wert,Bits) Long-Rotate-Left
Var=ROL&(Wert,Bits) Word-Rotate-Left
Var=ROL|(Wert,Bits) Byte-Rotate-Left
```

Rotiert den Inhalt von 'Wert' um die Anzahl 'Bits' nach links. Das jeweils rechts frei werdende Bit wird dabei mit dem jeweils links herausgeschobenen Bit gefüllt.

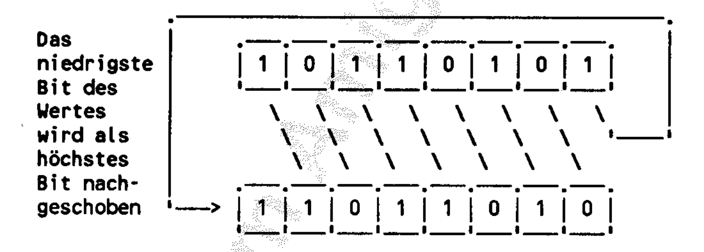


Bei Angabe von & hinter ROL werden nur die ersten 16 Bit (LO-Word) von Wert rotiert, bei | nur die ersten 8 Bit (LO-Byte). Ist Wert bei ROL|() größer als 8 Bit, so werden als Ergebnis trotzdem nur die untersten 8 Bit der Operation geliefert. Dasselbe gilt für ROL&() (16 Bit), wobei dann jedoch das Bit 15 des Ergebnisses in die Bits 16-31 kopiert wird (vorzeichenbehaftet).

ROR() — Bits rechts rotieren

```
Var=ROR(Wert,Bits) Long-Rotate-Right
Var=ROR&(Wert,Bits) Word-Rotate-Right
Var=ROR|(Wert,Bits) Byte-Rotate-Right
```

Rotiert den Inhalt von 'Wert' um die Anzahl 'Bits' nach rechts. Das jeweils links freiwerdende Bit wird dabei mit dem jeweils rechts herausgeschobenen Bit gefüllt.



Bei Angabe von & hinter ROR werden nur die ersten 16 Bit (LO-Word) von Wert rotiert, bei | nur die ersten 8 Bit (LO-Byte). Ist Wert bei ROR|() größer als 8 Bit, so werden als Ergebnis trotzdem nur die untersten 8 Bit der Operation geliefert. Dasselbe gilt für ROR&() (16 Bit), wobei dann jedoch das Bit 15 des Ergebnisses in die Bits 16-31 kopiert wird (vorzeichenbehaftet).

SWAP() — HI- und LO-Word vertauschen

```
Var=SWAP(Wert)
```

- Funktion - (siehe SWAP als Befehl)

Vertauscht das LO-Word (Bits 0-15) von 'Wert' mit dessen HI-Word (Bits 16-31). 'Wert' wird dabei grundsätzlich als 32-Bit-Integerwert interpretiert. Wird 'Wert' als Realwert übergeben, werden evtl. vorhandene Nachkommastellen vor Ausführung der Operation integriert (siehe INT()). Kleinere Wert-Formate als 32 Bit werden auf Long erweitert.

WORD() — Wert auf 32 Bit erweitern

```
Var=WORD(Wert)
```

Erweitert 'Wert' arithmetisch (vorzeichenbehaftet) auf 32 Bit. Das Bit 15 von 'Wert' wird dabei in die obersten 16 Bits des Ergebnisses kopiert. Ist also Bit 15 (16. von links) von 'Wert' gesetzt, so ist das Ergebnis von WORD(Wert) negativ.

XOR() — eXclusivOR-Funktion

```
Var=XOR(Wert1,Wert2)
```

Verknüpft logisch die beiden angegebenen Werte im XOR-Modus und liefert das Integer-Ergebnis. Weiteres siehe unter AND().

8.7 Zufallswert-Erzeugung

RAND() — 16-Bit-Integer-Zufallszahl

```
Var=RAND(n)
```

Übergibt eine vorzeichenlose 16-Bit-Integer-Zufallszahl aus dem Zahlenbereich 0 (inkl.) und n (exkl.). Größere Werte für n als 65535 werden durch n MOD 65535 auf den zulässigen Bereich umgerechnet.

RANDOM() — 32-Bit-Integer-Zufallszahl

```
Var=RANDOM(n)
```

Übergibt eine 32-Bit-Integer-Zufallszahl aus dem Integer-Zahlenbereich 0 (inkl.) und n (exkl.), wobei n auch negativ sein kann.

RANDOMIZE { RA } — Zufallszahlengenerator initialisieren

```
RANDOMIZE [(Start)]
```

Initialisiert den Zufallszahlengenerator. Bei Verwendung des optionalen Parameters Start wird der Generator mit diesem Wert gestartet. Bei mehrfacher Verwendung desselben Startwertes beginnt immer dieselbe Zufallszahlenfolge mit 'Start' als erstem Wert. Möchten Sie denselben Startwert intialisieren, der bei Systemstart gültig war, so kann die optionale Start-Klammer weggelassen oder RANDOMIZE 0 verwendet werden.

RND() Dezimalstellen-Zufallszahl

```
Var=RND [(Arg)]
```

Es wird ein 13stelliger Zufallswert im Bereich zwischen 0 (inkl.) und 1 (exkl.) geliefert. Die gesamte Klammer ist optional und kann vernachlässigt werden. Wenn sie verwendet wird, so gilt 'Arg' als Scheinargument ohne Bedeutung. Beispiel:

```
Print RND(0)*10+3     ! Liefert zufällige Realzahl
                      ! im Bereich von 3 bis 13
```

9. Grafik

Die Grafikbefehle der GFA-BASIC sind zu großen Teilen noch vom ATARI ST übernommen worden. Allerdings hat sich hier gezeigt, daß doch ein Großteil zu sehr auf die speziellen Bedürfnisse zugeschnitten war und so nicht übernommen werden konnte.

Andererseits - und dies sei denjenigen zum Trost, die verlorenen Befehlen hinterhertrauern - sind neue und weit mächtigere Befehle hinzugekommen, da die Bereiche BOB (BlitterOBjects = Grafikobjekte) und Screen beim ATARI niemandem bekannt waren.

9.1 Grafikdefinitionen

BOUNDARY { BOU } — P-Grafikumrandung an/aus

`BOUNDARY Flag` *Version 3.0*

Schaltet die Umrandung von P-Grafikobjekten (PBOX, PCIRCLE, POLYFILL etc.) an (Flag = 1) oder aus (Flag = 0).

Beispiel: Diese Prozedur eignet sich zur Erzeugung der unterschiedlichsten geometrischen Figuren. Spielen Sie ein wenig mit den Einstellungsparametern und den Punktangaben der POLY-xxx-Befehle. Das Koordinatenfeld wurde vorsorglich auf maximal 60 Punkte dimensioniert, um Veränderungen der Schrittweite zulassen zu können. Ein rechtwinkliges Dreieck läßt sich z.B. dadurch erzeugen, indem daß Sie die Schrittweite STEP von 120 auf 90 ändern. Ein exzentrisches Dreieck entsteht, wenn Sie STEP kleiner 90 werden lassen. Ein gleichmäßiges Sechseck ent-

steht, indem Sie STEP 60 einstellen und gleichzeitig die Punktangaben bei POLYLINE auf 7 sowie bei POLYFILL auf 6 Punkte ändern.

```
DEFMARK ,4                          ! Marker einstellen
DEFFILL ,2,4                        ! DEFILL grau
FOR i%=0 TO 360 STEP 20             ! Einmal rundherum in 20-Grad-Steps
  BOUNDARY 0                        ! Rahmen aus
  @ctri(1,120,110,100,i%)
  @ctri(2,120,110,60,i%)
  BOUNDARY 1                        ! Rahmen an
  @ctri(2,520,110,60,i%)
NEXT i%
'
PROCEDURE ctri(mod%,xp%,yp%,rd%,wi%) ! Für Hires
  ' Produziert ein Dreieck, das in einem beliebigen Winkel
  ' und mit beliebiger Größe dargestellt werden kann.
  ' Mod%    = Darstellungsmodus
  '           1 = Nur Linie
  '           2 = Gefüllte Dreiecksfläche
  ' Möchten Sie nur die Eck-Koordinaten ermitteln, ohne daß
  ' die Figur gezeichnet wird, so übergeben Sie Mod% = 0.
  ' Die Eck-Koordinaten können Sie dann nach Rückkehr aus
  ' Px%(0)/Py%(0) bis Px%(2)/Py%(2) auslesen.
  '
  ' Xp%/Yp% = Koordinaten des Mittelpunktes (Drehpunktes)
  ' Rd%     = Umkreisradius
  ' Wi%     = Neigungswinkel
  '
  LOCAL j%,i%,yt%                   ! Lokale Variablen
  yt%=1                             ! Y-Auflösungsteiler
  ERASE px%()                       ! POLY-X-Feld löschen
  ERASE py%()                       ! POLY-Y-Feld löschen
  DIM px%(4),py%(4)                 ! POLY-Felder dimensionieren
  FOR i%=-wi% TO -wi%+360 STEP 120 ! Einmal rundum
    px%(j%)=xp%+(SIN(i%*PI/180)*rd%+0.5)     ! Koordinaten...
    py%(j%)=yp%+(COS(i%*PI/180)*rd%/yt%+0.5) ! ...berechnen
    INC j%                          ! Ecken-Zähler +1
  NEXT i%                           ! Nächste Ecke
  IF mod%=1                         ! Nur Linie?
    POLYLINE 4,px%(),py%()          ! Dann POLYLINE
  ENDIF
  IF mod%=2                         ! Gefüllte Figur?
    POLYFILL 3,px%(),py%()          ! Dann POLYFILL
  ENDIF
RETURN
```

COLOR { C } — Linienfarbe bestimmen

```
COLOR Front, Back, OutLine
```

Bestimmt das Farb-Register, aus dem linien- und punktezeichnende Grafikobjekte (LINE, PLOT, DRAW, CIRCLE, BOX etc.) ihre Farben beziehen. Dabei gibt Front die Farbnummer an, mit der Linien gezeichnet werden, Back, mit welcher Farbe der Hintergrund gezeichnet werden soll, und OutLine enthält die Farbnummer, mit der Objekte, die ein P am Anfang des Befehls haben, umrandet werden.

Der Auswahlbereich für die Farbnummern hängt dabei ganz von der Screen-Auflösung ab. Er kann maximal zwischen 0 und 63 liegen, wird aber bei höheren Auflösungen oder weniger Bit-Planes heruntergesetzt.

Allgemein gilt:

1 BitPlane	2 Farben
2 BitPlanes	4 Farben
3 BitPlanes	8 Farben
4 BitPlanes	16 Farben
5 BitPlanes	32 Farben
6 BitPlanes	64 Farben

Allerdings gelten folgende Einschränkungen:

- Im Hires-Modus (also 640 x 256 Punkte) können maximal 4 BitPlanes, also 16 Farben angesprochen werden.
- Es ist nicht möglich, mehr als 5 BitPlanes ohne besondere Tricks zu verwalten.

Daher fragt man sich, wozu die höheren Farbnummern von 32 bis 63 gebraucht werden. Diese werden für zwei besondere Grafikmodi des Amiga benötigt. Und zwar ist dies einmal der EXTRA_HALFBRITE- und der HAM-Modus, zu denen Sie mehr unter OPENS lesen können.

Auflösung		BitPlanes	Farben
Hires	(640/256)	4	0 ... 16
Hires Lace	(640/512)	4	0 ... 16
Lowres	(320/256)	5	0 ... 31
Lowres Lace	(320/512)	5	0 ... 31
HAM	(320/256)	6	0 ... 63 (aber 4096 wirklich)
HAM Lace	(320/512)	6	0 ... 63 (aber 4096 wirklich)
EHB	(320/256)	6	0 ... 63 (32 mit 2 Helligkeiten)
EHB Lace	(320/512)	6	0 ... 63 (32 mit 2 Helligkeiten)

Bei den Auflösungen gibt es Farben, die bei Systemstart voreingestellt sind. Der Inhalt der Register kann mit dem Befehl SETCOLOR bestimmt werden. Ein Beispiel finden Sie unter SETCOLOR.

DEFFILL { DEFF } — Füllmuster bestimmen

```
DEFFILL [Farbe],[Stil],[Muster]
DEFFILL [Farbe],Muster$
```

Legt Füllfarbe, Füllstil und Füllmuster für P-Grafikbefehle und FILL fest bzw. ermöglicht eigene Muster-Definitionen. Parameter Farbe siehe unter COLOR.

Stil	Muster
0 = Hintergrundfarbe	Entfällt
1 = Objektfarbe	Entfällt
2 = Punktiert	2 bis 24
3 = Schraffiert	1 bis 12
4 = Selbstdefiniert	Amiga- bzw. Benutzer-Muster

Parameter, die unverändert bleiben sollen, können ausgelassen werden (jedoch nicht die dazugehörigen Trennkommas). Mit der Variante DEFFILL Farbe,Muster$ läßt sich ein eigenes Füllmuster einrichten. Dazu ist in Muster$ ein 16 (bzw. 32/64 siehe unten) Words langer String zu übergeben. Dieser beinhaltet 16 Words (bzw. 32/64 siehe unten) im MKI$-Format, die der Reihe nach die Bit-Muster der 16 Musterzeilen enthalten. Beispiel 1:

```
F.muster$=Mki$(&X1111111111111111)
F.muster$=F.muster$+Mki$(&X1000000000000000)
F.muster$=F.muster$+Mki$(&X1000000110000000)
F.muster$=F.muster$+Mki$(&X1000001001000000)
F.muster$=F.muster$+Mki$(&X1000010000100000)
F.muster$=F.muster$+Mki$(&X1000100000010000)
F.muster$=F.muster$+Mki$(&X1001000110001000)
F.muster$=F.muster$+Mki$(&X1010001111000100)
F.muster$=F.muster$+Mki$(&X1001000110001000)
F.muster$=F.muster$+Mki$(&X1000100000010000)
F.muster$=F.muster$+Mki$(&X1000010000100000)
F.muster$=F.muster$+Mki$(&X1000001001000000)
F.muster$=F.muster$+Mki$(&X1000000110000000)
F.muster$=F.muster$+Mki$(&X1011111111111110)
F.muster$=F.muster$+Mki$(&X1001111111111000)
F.muster$=F.muster$+Mki$(&X1000000000000000)
Deffill 1,F.muster$
Pbox 100,100,200,200
```

Es kann für jede mögliche Bit-Plane jeweils ein 16 Word-Block in Muster$ übergeben werden. D.h., in den Words 1 bis 16 steht dann das Bit-Muster für die Plane 1, in den Words 17 bis 32 das zusätzliche Bit-Muster für Plane 2, und in den Words 33 bis 48 steht das zusätzliche Bit-Muster für Plane 3 usw.

Beispiel 2:

```
GRAPHMODE 2
FOR i%=0 TO 7            !-------.
  DEFFILL i%,3,i%                ! Irgendein
  PCIRCLE 16,16,16               ! Bit-Muster
  DEFFILL i%+2,3,i%+2            ! erzeugen
  PCIRCLE 48,16,16       !-------'
NEXT i%
@dfill3(7,7,df$)         ! V3.0-Aufruf
DEFFILL 1,df$            ! Neues Füllmuster..
PBOX 15,64,192,192       ! ...zeigen
GRAPHMODE 3              ! XOR-Modus
@dfill3(42,7,df$)        ! V3.0-Aufruf
```

```
DEFFILL 1,df$                  ! Neues Füllmuster..
PBOX 105,14,192,112            ! Zeigen (mit Überlappung)
'
PROCEDURE dfill3(d.x%,d.y%,VAR d.ff$)    ! nur für V3.0
  ' Kopiert einen beliebigen 16*16-Bildschirmausschnitt im
  ' DEFFILL-Format in eine String-Variable (Hires/Midres/Lowres)
  ' D.x%  = Linke Quell-X-Koordinate
  ' D.y%  = Obere Quell-Y-Koordinate
  ' D.ff$ = VAR-String-Variable, welche nach Abschluß
  '         die Füllmusterdaten enthält
  LOCAL d.fr$,dc1%,dc2%,d.f$,xb%           ! Lokale Variablen
  GET d.x%,d.y%,d.x%+15,d.y%+15,d.fr$      ! Ausschnitt speichern
  d.f$=RIGHT$(d.fr$,32)                    ! Muster übertragen
  d.ff$=d.f$                               ! Muster-Rückgabe
RETURN
```

DEFLINE { DE } — Linien-Modi bestimmen

```
DEFLINE [Stil]
```

Mit diesem Befehl kann das bei linienzeichnenden Befehlen (BOX, LINE, DRAW etc.) zu verwendende Linienmuster festgelegt werden. ACHTUNG! Bei den Befehlen CIRCLE und ELLIPSE wird das hier definierte Linienmuster nicht verwendet.

Stil:

0 = Linie in Hintergrundfarbe
1 = Durchgezogene Linie
2 = Gestrichelte Linie 1
3 = Gepunktete Linie
4 = Punkt-Strich-Linie
5 = Gestrichelte Linie 2
6 = Strich-Punkt-Punkt-Linie
-1 bis -32767 Benutzerstil

Der selbstdefinierte Linienstil setzt sich aus einem 15-Bit-Wert zusammen, wobei jedes gesetzte Bit einem Punkt in der Linie entspricht. Diese Zahl muß als Minuswert übergeben werden. Die Linie setzt sich dann aus dem Vielfachen dieser 15 Bits zu-

sammen. Die Stil-Veränderungen wirken sich nur bei Liniendicke 1 aus. Alle anderen Liniendicken werden als Voll-Linie gezeichnet.

0 = Eckig
1 = Pfeilförmig
2 = Rund

Beispiel:

```
FOR i%=1 TO 6
  DEFLINE i%
  LINE 200,60+i%*16,300,60+i%*16
  PRINT AT(40,8+i%*2);i%
NEXT i%
DEFLINE -18149       ! -18149 = -&X1000110111100101
FOR i%=1 TO 90 STEP 4
  BOX 10+i%,10+i%,190-i%,190-i%
NEXT i%
```

DEFMOUSE { DEFM } — Mausform bestimmen

```
DEFMOUSE Form
DEFMOUSE Maus$
```

Aufruf von selbstdefinierten oder systemeigenen Mausformen. Dieser Mauszeiger kann genau wie bei der allgemeinen Programmierung für jedes Window separat eingestellt werden. Dadurch kann der Benutzer genau erkennen, welches Window gerade aktiv ist. Wenn z.B. im Eingabewindow der Mauszeiger die Gestalt eines Pfeils und im Status Window die einer Sanduhr hat, weiß man, daß die Eingabe zwar möglich ist, der Statusbericht aber noch abgewartet werden muß.

Form (vordefiniert):

0 = Pfeil
1 = X-Klammer (Text-Cursor)
2 = Sanduhr
3 = Zeigende Hand
4 = Offene Hand
5 = Fadenkreuz fein
6 = Fadenkreuz grob
7 = Fadenkreuz umrandet

Statt durch Form kann durch Maus$ eine String-Variable angegeben werden, deren Inhalt im MKI$-Format die Mausform definiert.

Word 1

X-Koordinate des Aktionspunktes.

Word 2

Y-Koordinate des Aktionspunktes. Auf den Aktionspunkt werden anschließend alle Mausaktionen (z.B. Mousex, Mousey) bezogen.

Word 3

Immer MKI$(1).

Word 4

Maskenfarbe (Hintergrund des Mausbildes):

```
weiß = MKI$(0)/schwarz = MKI$(1)
```

Word 5

Cursor-Farbe (Mausbild).

Word 6 bis 21

16-Bit-Musterzeilen der Mausmaske.

Word 22 bis 37

16-Bit-Musterzeilen des Mausbildes.

Die Definition einer Mausform ist beim Amiga vierfarbig möglich.

Beispiel:

```
OPENW 0                ' Ausgabe-Window öffnen
PAUSE 20               ' Zeit zum Anklicken geben
FOR i=0 TO 8
  DEFMOUSE i           ' neues Maus-Image definieren
  PAUSE 40             ' Zeit zur Anzeige geben
NEXT i
```

GRAPHMODE { GR } — Grafikmodus bestimmen

```
GRAPHMODE Modus
```

Modus bestimmt den Operationsmodus, mit welchem Grafikausgaben in den bestehenden Hintergrund eingesetzt werden.

0 = JAM1 (transparent)

Das verwendete Grafik-Element (PBOX, LINE etc) wird vollständig gezeichnet. Allerdings beschränkt sich die Textausgabe auf die Linien, die zur Darstellung benötigt werden. Der Raum um den Text wird nicht verändert.

```
Neuer Punkt = Farbmaske des neuen Punktes OR neuer Punkt
```

1 = JAM2 (replace)

Das verwendete Grafik-Element (PBOX, LINE etc.) wird vollflächig dargestellt. Alles, was sich darunter befindet, wird davon überdeckt und ersetzt. Bei der Textausgabe wird auch der Hintergrund, auf den man den Text schreibt, mit der Hintergrundfarbe überschrieben.

```
Neuer Punkt = Farbmaske des neuen Punktes AND neuer Punkt
```

2 = COMPLEMENT (invertiert)

Es werden alle Bildpunkte gesetzt, die vorher nicht gesetzt waren und umgekehrt. Dies geschieht aber nur an den Stellen, an

denen auch eine Grafik ausgegeben wird. COMPLEMENT arbeitet in der Beziehung wie auch JAM1.

4 = INVERSEVID (invertiert)

Bei diesem Ausgabemodus wird z.B. der auszugebende Text invertiert dargestellt. Das heißt, dort, wo Text stehen soll, wird der Grafikbildschirm gelöscht. Und dort, wo im Normalfall Hintergrund erscheint, ist jetzt Dunkelheit (der Amiga setzt dort die Punkte).

Alle diese Zeichenmodi lassen sich untereinander verknüpfen! Sie können durch Addieren der Kennzahlen zweier Zeichenmodi einen neuen kreieren. Allerdings gibt es natürlich auch hier Grenzen. Deshalb wurde z.B. JAM1 mit der Null belegt. Sie können, so oft Sie wollen, eine Null zu einem anderen Zeichenmodus addieren, und es wird sich nichts ändern. Andererseits sind Kombinationen wie JAM2 und COMPLEMENT durchaus anwendbar und erzeugen interessante Effekte.

Die vier Grund-Modi zeigt die folgende Grafik:

Beispiel:

```
DEFFILL ,2,5
PBOX 10,10,200,90
BOX 8,8,202,124
TEXT 36,118,"G R A P H M O D E"
DEFFILL ,2,4
FOR i%=1 TO 4
  GRAPHMODE 2^(i%-2)
  PBOX i%*50-35,22,i%*50-20,160
  GRAPHMODE 0
  TEXT i%*50-18,158,INT(2^(i%-2))
NEXT i%
```

SETCOLOR { SET } Hardware-Farbregister einstellen

```
SETCOLOR Reg,Rot,Grün,Blau
SETCOLOR Reg,Mischwert
```

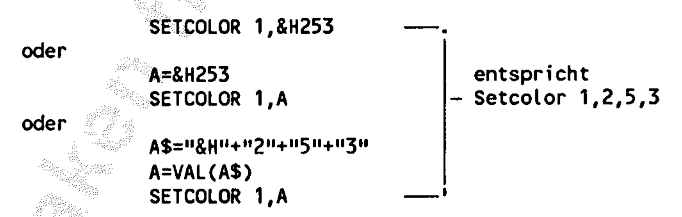
Die Farbe des mit Reg angegebenen Farbregisters der aktiven Screen kann entweder durch die Farbanteile Rot, Grün und Blau (jeweils 0 bis 15) oder durch einen Mischwert (0 bis 4095) definiert werden.

```
Mischwert = (Rotanteil*256)+(Grünanteil*16)+Blauanteil
```

Bei verschiedenen Gelegenheiten ist es von Nutzen, die Registereinträge nicht mit drei unabhängigen Parametern vorzunehmen, sondern einen Gesamt-Mischungswert zu übergeben. GFA-BASIC unterscheidet, ob ein oder mehrere Farbparameter übergeben wurden. Wird nur der Parameter Mischwert übergeben, setzt sich die Farbmischung nach folgendem Muster zusammen:

Soll z.B. ein Blauanteil von 3, ein Rotanteil von 2 und ein Grünanteil von 5 übergeben werden, errechnet sich der Wert, indem man den Rotanteil mit 256 (2^8) multipliziert, den Grünanteil mit 16 (2^4) und den Blauanteil mit 1 (2^0). In diesem Beispiel müßte der Wert 595 übergeben werden, um die gewünschte Farbmischung zu erhalten.

Durch die Möglichkeit, mit Hexadezimalzahlen zu arbeiten, kann man diese Berechnung folgendermaßen vereinfachen:



So lassen sich 4096 verschiedene Farben (3 Grund-Farben mit je 16 Abstufungen = 16*16*16 = 4096 Farben) erzeugen.

Die Bits 0 - 7 von Mischwert bestimmen dabei den Rotanteil, die Bits 8 - 15 den Grünanteil und die Bits 16 - 23 den Blauanteil der Farbe.

Bei Verwendung der Standard-Ausgabe auf der Workbench mit dem GFA-Ausgabe-Fenster haben die vier Farbregister eine festgelegte Zuordnung:

Register 0 = Hintergrund-Farbe (Blau)
Register 1 = PRINT-Ausgabetext-Farbe (Weiß)
Register 2 = Block-Farbe (Schwarz)
Register 3 = Signal-Farbe für Warnungen (Rot)

Leider ist der Goethesche Farbkreis hier nicht anwendbar, da hier nicht die drei Elementarfarben Rot, Blau und Gelb gemischt werden. Um sich bei der Farbwahl zu orientieren, müßte aus Grün die Farbe Blau subtrahiert werden, um Gelb zu erhalten. Trotzdem kann man sich seiner als allgemeine Richtlinie bedienen:

```
           ROT
VIOLETT         ORANGE
         WEISS
BLAU               GELB
         GRÜN
```

Zur Farbphysik:

Es existieren drei Elementarfarben : ROT, BLAU, GELB.

Die 1:1-Mischung zweier dieser Farben ergibt die Komplementärfarbe (Gegenfarbe) der nicht-beteiligten dritten Farbe.

1:1 Mischung von ROT und BLAU ergibt VIOLETT (= Komplement zu GELB).

1:1 Mischung von ROT und GELB ergibt ORANGE (= Komplement zu BLAU).

1:1 Mischung von BLAU und GELB ergibt GRÜN (= Komplement zu ROT).

1:1:1 Mischung von ROT, BLAU und GELB ergibt WEISS.

Ist keine der Elementarfarben beteiligt, ergibt sich die "Nichtfarbe" SCHWARZ. Beispiel:

```
DIM c%(15)                  ! DIM Farb-Speicher
SETCOLOR 0,0                ! Hintergrund schwarz
FOR i%=1 TO 15              ! 15 Register (außer 0) >------.
  IF i%<8                   ! 7 Werte von weiß zu grün     |
    c%(i%)=i%*16            ! Grünanteil erhöhen           |
  ELSE                      ! 8 Werte von gelb zu rot      |
    c%(i%)=7*256+(15-i%)*16 ! Grünanteil vermindern        |
  ENDIF                                                    |
  SETCOLOR i%,c%(i%)        ! Farbe setzen                 |
NEXT i%                     ! Nächstes Register <----------'
FOR j%=0 TO 4               ! 5 mal >--------------------------.
  RESTORE                   ! Data-Zeiger setzen               |
  FOR i%=1 TO 15            ! 15 Farben  >-----------------.   |
    READ a%                 ! COLOR-Register holen         |   |
    COLOR a%                ! BOX- und CIRCLE-Farbe setzen |   |
    BOX 200+j%*20+i%,20+j%*20+i%,400-j%*20-i%,220-j%*20-i% |   |
    CIRCLE 300,120,j%*15+i%*10                             |   |
    CIRCLE 300,120,j%*15+i%*10+5                           |   |
  NEXT i%                   ! Nächste Farbe <--------------'   |
NEXT j%                     ! Nächster Offset <----------------'
DO                          ! Endlos-Schleife >------------.
  b%=c%(15)                 !--.                           |
  FOR i%=14 DOWNTO 1           |                           |
    c%(i%+1)=c%(i%)            |- Farben rotieren          |
  NEXT i%                      |                           |
  c%(1)=b%                  !--'                           |
  FOR i%=1 TO 15            ! 15 Register >-----------.    |
    FOR j%=1 TO 100         !    Kleine...            |    |
    NEXT j%                 !    ...Pause             |    |
    SETCOLOR i%,c%(i%)      ! Neue Farbe  setzen      |    |
  NEXT i%                   ! Nächstes Register <-----'    |
LOOP                        ! <----------------------------'
DATA 2,3,6,4,7,5,8,9,10,11,14,12,15,13,1, Umrechnungstabelle
```

Dieses Beispiel produziert eine Farbspielerei, die für Interessierte leicht zur Meditationshilfe (für gestreßte Programmierer) oder zur Bio-Feedback-Methode "zweckentfremdet" werden kann. Die Farben pulsieren mit einer Frequenz von ca. 58 Zyklen pro Minute (ca. optimale Pulsfrequenz bei Entspannung). Die Frequenz läßt sich leicht mit der Pausen-Schleife im letzten Block verändern.

9.2 Objektgrafikbefehle

BOX, PBOX { BO, PB } **Rechteck zeichnen**

```
[P]BOX X_links,Y_oben,X_rechts,Y_unten
```

X_links/Y_oben und X_rechts/Y_unten bezeichnen die diagonal gegenüberliegenden Ecken eines Rechtecks, das entweder als Linienzug (BOX) oder mit dem aktuellen DEFFILL-Füllmuster (PBOX) gezeichnet wird.

Beispiel: Wenn es darum geht, ein größeres Raster mit PBOX zu füllen (z.B. in einer Lupe), sollte man - wenn irgend möglich - versuchen, die PBOX durch eine PUT-Fläche zu ersetzen. Das geht erheblich schneller. Bei kleinen Rastern fällt der Geschwindigkeitsunterschied nicht besonders auf, bei größeren Rastern ist es dagegen schon ein merklicher Unterschied, ob das Raster in 2 oder in 4 Sekunden gefüllt wird. Um das zu demonstrieren, folgt eine kleine Routine, die es ermöglicht, einen beliebigen Bildausschnitt zu vergrößern und/oder ihn für eine spätere Verwendung (raster-indiziert) in einem Feld zwischenzuspeichern.

Hier ist eine kleine Bedienungsanleitung nötig. Sie können den Bildbereich, in dem der vergrößerte Ausschnitt dargestellt werden soll, beliebig bestimmen. Außerdem kann die Rastergröße und die Breite und Höhe eines vergrößerten Rasterpunktes angegeben werden. Es ist ungünstig, wenn sich der Wiedergabe-Ausschnitt mit dem zu vergrößernden Ausschnitt überschneidet. Der Wiedergabe-Ausschnitt wird - sollten seine Eck-Koordinaten außerhalb des Bildschirms liegen - von der Routine auf den sichtbaren Bildschirm begrenzt.

Ist die Routine aufgerufen, so wird ggf. der vergrößerte Ausschnitt gezeichnet und die Routine wartet dann darauf, daß mit der rechten Maustaste der Lupeninhalt verändert wird. Verlassen wird die Lupe zu jedem beliebigen Zeitpunkt mit Druck auf die rechte Maustaste. Daran anschließend wird der Lupenhintergrund selbständig restauriert.

Der Lupeninhalt wird verändert, indem mit der linken Maustaste der gewünschte Lupenpunkt angeklickt wird. Der Punkt verändert seine Farbe. Diese Farbe behält er auch bei Mausbewegungen solange bei, bis die Maustaste wieder losgelassen wird. Bei jedem Mausklick auf einen Lupenpunkt wird sein Farbindex um 1 vermindert. Klicken Sie also auf der Workbench viermal auf denselben Punkt, nimmt er nacheinander alle 4 möglichen Farben an. Es ist deshalb ratsam, bei einem Punkt-Klick die Maustaste gedrückt zu halten, um zu sehen, welche Farbe sich ergibt. Solange Sie die Taste gedrückt halten, können Sie mit dieser Farbe innerhalb der Lupe zeichnen. Wird die Taste losgelassen, wird der Farbindex, auf dem sich die Maus zum Zeitpunkt des Klicks befindet um 1 vermindert. Haben Sie als Flag 1 angegeben, wird gleichzeitig auch der originale Bildschirmbereich entsprechend der Lupenänderung verändert. Nach Rückkehr aus der Prozedur ist der aktuelle Lupeninhalt in dem Feld Larr%(Rasterbreite,Rasterhöhe) gespeichert und kann damit auch außerhalb der Prozedur verwendet werden.

Soll die Lupe nicht dargestellt werden, geben Sie als Flag 2 an. In diesem Fall wird nur das Feld Larr%() mit dem angegebenen Bildausschnitt gefüllt. Der Feldindex entspricht der Lage der Punkte im Raster. Die Indizierung beginnt mit Null. Z.B. Larr%(0,0) enthält den Farbwert des 1. Punktes links oben in der Ausschnittecke und z.B. Larr%(12,6) den Farbwert des Rasterpunktes mit den Rasterkoordinaten 13. Punkt von links/7. Punkt von oben. Grundsätzlich ist zu bedenken, daß das Feld - abhängig von der Rastergröße - eine ganze Menge Speicherplatz verbrauchen kann. Ein Raster von z.B. 50*50 Punkten benötigt (50*50*4) 10000 Bytes.

Weitere Informationen finden Sie unten in der Beschreibung der Prozedur.

```
' Vorbereitung für beide Anwendungsbeispiele
' ----------------------------------------------
FOR i%=0 TO 100                   ! 100 mal
  DEFFILL RANDOM(MAX(1,(2^(2-2))^2-1))+1,2,RANDOM(22)
  x%=RANDOM(590)                  ! PBOX mit zufälliger...
  y%=RANDOM(200)                  ! ...Farbe, Muster und...
  PBOX x%,y%,x%+50,y%+50          ! ...Position zeichnen
```

```
NEXT i%
'
' Anwendungsbeispiel 1:
' ----------------------
PRINT "Lupe mit rechter Maustaste aufrufen"
DO                              ! Endlos-Schleife
  GRAPHMODE 3                   ! XOR-Modus für Flimmerbox
  MOUSE xx%,yy%,k%              ! Maus-Status holen
  xx%=MAX(30,xx%)              ! Box-X-Position begrenzen
  yy%=MAX(30,yy%)              ! Box-Y-Position begrenzen
  BOX xx%-1,yy%-1,xx%+20,yy%+20 ! Flimmerbox...
  BOX xx%-1,yy%-1,xx%+20,yy%+20 ! ...zeichnen
  IF MOUSEK=2                   ! Rechte Maustaste gedrückt?
    GRAPHMODE 1                 ! REPLACE-Modus an
    lupe(xx%-20*6,yy%-20*6,20,20,6,6,1) ! Aufruf
  ENDIF
LOOP
'
' Anwendungsbeispiel 2 (vorher Beispiel 1 löschen):
' --------------------------------------------------
PRINT "Maushintergrund (20*20) in Feld Larr%() einlesen."
PRINT "beliebige Maustaste drücken"
REPEAT                          ! Auf...
UNTIL MOUSEK                    ! Mausklick warten
@lupe(MOUSEX-20*6,MOUSEY-20*6,20,20,6,6,2) ! Lupe aufrufen
CLS                             ! Bildschirm löschen
FOR i%=0 TO 20-1                ! 20 Zeilen
  FOR j%=0 TO 20-1              ! 20 Spalten
    COLOR larr%(i%,j%)          ! Punktfarbe setzen
    PLOT 100+i%,100+j%          ! Punkt zeichnen
  NEXT j%
NEXT i%
'
PROCEDURE lupe(xp%,yp%,br%,ho%,sx%,sy%,flg%)
  ' Für Hires/Midres/Lowres
  ' Xp% = X-Koordinate der Lupenbox
  ' Yp% = Y-Koordinate der Lupenbox
  ' Br% = Breite des zu vergrößernden Rasters
  ' Ho% = Höhe des zu vergrößernden Rasters
  ' Sx% = Breite eines Lupen-Rasterpunktes
  ' Sy% = Höhe eines Lupen-Rasterpunktes
  ' Flg%= Flag
  '        0 = Lupe wird gezeichnet und kann verändert
  '            werden. Der Original-Ausschnitt bleibt
  '            dabei unverändert.
  '        1 = Lupe wird gezeichnet und kann verändert
  '            werden. Der Original-Ausschnitt wird
  '            entsprechend der Lupenänderungen ebenfalls
  '            verändert.
  '        2 = Es wird nur das Rasterfeld mit den Punktfarben
  '            des Original-Rasters gefüllt. Die Lupe wird
  '            nicht gezeichnet.
  ' Das Rasterfeld Larr%() wird auch bei Modus 0 und 1 gefüllt
```

```
' und kann nach Rückkehr aus der Routine ausgewertet werden.
'
LOCAL i%,j%,lb$,mx%,my%,mk%,xp2%,yp2%,pkt%
LOCAL pnt%,mxx%,myy%,pxr%,pyr%
sx%=MIN(sx%,16)                  ! Max. Punktbreite = 16
xp%=MIN(639,MAX(xp%-10,0))     ! Koordinaten der...
yp%=MIN(255,MAX(yp%-10,0))     ! ...Wiedergabe-Box...
xp2%=MIN(639,MAX(xp%+br%*sx%+9,0)) ! ...auf Bildschirm...
yp2%=MIN(255,MAX(yp%+ho%*sy%+9,0)) ! ...begrenzen
GET xp%,yp%,xp2%,yp2%,lb$        ! Hintergrund sichern
ERASE larr%()                    ! Rasterfeld löschen
DIM bl$((2^(2-2))^2),larr%(br%,ho%) ! DIM PUT-
'                                ! und Rasterfeld
IF flg%<>2                       ! Lupe zeichnen?
  FOR i%=0 TO (2^(2-2))^2  ! Alle möglichen Farben
    DEFFILL i%,2,8               ! Füllmusterfarbe setzen
    pxr%=xp%+4+MIN(br%*sx%,18)   ! Breite u. Höhe der...
    pyr%=yp%+4+MIN(ho%*sy%,18)   ! ...PBOX ermitteln
    PBOX xp%+4,yp%+4,pxr%,pyr%   ! PBOX vollfarbig zeichnen
    GET xp%+5,yp%+5,xp%+5+15,yp%+5+15,bl$(i%) ! PUT-Box...
    '                            ! ..speichern (PBOX-Ersatz)
    MID$(bl$(i%),1,4)=MKI$(MAX(1,sx%-1))+MKI$(MAX(1,sy%-1))
    ' PUT-Header an gewünschte Punktgröße anpassen
  NEXT i%
  DEFFILL 0,0,0                  ! DEFFILL weiß
  COLOR 1                        ! COLOR für Rahmenbox
  PBOX xp%,yp%,xp2%,yp2%         ! Lupenhintergrund löschen
  BOX xp%,yp%,xp2%,yp2%          ! Lupenrahmen zeichnen
  BOX xp%+4,yp%+4,xp2%-4,yp2%-4  ! Lupenrahmen zeichnen
ENDIF
MOUSE mx%,my%,mk%                ! Aktuelle Mauskoordinate
FOR i%=0 TO br%-1                ! Raster-X-Index
  FOR j%=0 TO ho%-1              ! Raster-Y-Index
    pnt%=POINT(mx%+i%,my%+j%)    ! Punktfarbe ermitteln
    IF pnt%                      ! Punkt gesetzt?
      larr%(i%,j%)=pnt%          ! Punktfarbe in Feld schreiben
      IF flg%<>2                 ! Lupe soll gezeichnet werden?
        PUT xp%+5+i%*sx%,yp%+5+j%*sy%,bl$(pnt%) ! Lupenpunkt...
      ENDIF                                     ! ...zeichnen
    ENDIF
  NEXT j%
NEXT i%
IF flg%<>2                       ! Lupe ist gezeichnet?
  REPEAT                         ! Lupen-Schleife
    MOUSE mxx%,myy%,mk%          ! Maus-Status holen
    REPEAT                       ! warten...
    UNTIL MOUSEX<>mxx% OR MOUSEY<>myy% OR MOUSEK
    '                            ! ...auf Mausaktion
    IF mxx%>xp%+5 AND mxx%<xp%+4+br%*sx%   ! X-Maus in der Lupe?
      IF myy%>yp%+5 AND myy%<yp%+4+ho%*sy% ! Y-Maus in der Lupe?
        mxx%=INT((mxx%-(xp%+5))/sx%) ! Koordinaten auf...
        myy%=INT((myy%-(yp%+5))/sy%) ! ...Rasterindex umrechnen
        IF mk%=1                 ! Linke Maustaste gedrückt?
```

```
        IF flg%=1                ! Originalpunkt ändern?
          PLOT mx%+mxx%,my%+myy% ! Dann Punkt setzen
        ENDIF
        larr%(mxx%,myy%)=pkt%    ! Rasterfeld aktualisieren
        PUT xp%+5+mxx%*sx%,yp%+5+myy%*sy%,bl$(pkt%)
        '                        ! Lupenpunkt setzen
      ELSE                       ! linke Maustaste ist aus!
        pkt%=POINT(xp%+5+mxx%*sx%,yp%+5+myy%*sy%)-1
        '                        ! Punktfarbe um 1 vermindern
        IF pkt%<0                ! Farbindex < 0?
          pkt%=MAX(1,(2^(2-2))^2-1) ! Index auf Maximum
        ENDIF
        COLOR pkt%               ! Originalpunktfarbe setzen
      ENDIF
    ENDIF
  ENDIF
  UNTIL mk%=2                    ! Rechte Maustaste gedrückt?
  PAUSE 5                        ! Kleine Klick-Pause
 ENDIF
 ERASE bl$()                     ! PUT-Box-Feld löschen
 PUT xp%,yp%,lb$                 ! Hintergrund restaurieren
RETURN
```

CIRCLE, PCIRCLE { CI, PC } Kreis(bogen); zeichnen

```
[P]CIRCLE X_cent, Y_cent, Radius
```

"X_cent/Y_cent" bestimmen den Kreismittelpunkt. "Radius" ist der halbe Kreisdurchmesser.

CIRCLE zeichnet einen Kreis um den Mittelpunkt X_cent, Y_cent mit dem angegebenen Radius. Dieser darf momentan nur Werte bis zu 511 annehmen, weil der Blitter nicht in der Lage ist, größere Grafikflächen zu bearbeiten.

ELLIPSE, PELLIPSE { ELL, PE } Ellipse(nbogen) zeichnen

```
[P]ELLIPSE Xcent,Ycent,Xrad,Yrad
```

"Xrad" ist der Ellipsenradius in X-Richtung und "Yrad" der in Y-Richtung. "Xcent/Ycent" legt den Ellipsenmittelpunkt fest.

FILL { FI } Flächen mit Muster füllen

```
FILL Xpos,Ypos [,Farbe]
```

"Xpos/Ypos" gibt die Lage des Bildschirmpunktes an, von dem aus sich der Füllvorgang ausbreitet, bis er an geschlossene Grenzen stößt, also an Punkte mit einer anderen Farbe als der Punkt (Xpos,Ypos).

FILL ist ggf. mit Vorsicht zu genießen. Dies hat zwei Gründe. Der eine ist, daß das ausbreitende Füllmuster die kleinsten (1 Pixel) Schlupfwinkel entdeckt, durch die es sich weiter ausbreiten kann. Das führt manchmal dazu, daß Bereiche, die nicht gefüllt werden sollen, durch den FILL-Befehl zunichte gemacht werden und es dann oft unmöglich ist, diesen Bereich wieder vom unerwünschten Füllmuster zu säubern.

Auf der anderen Seite ist die Füll-Funktion besonders schnell, weil sie vom Blitter unterstützt wird, der in Windeseile Bereiche von bis zu 1024 x 1024 Punkten innerhalb nur einer Sekunde beschreiben kann. Diese Geschwindigkeit kann nur dadurch gebremst werden, daß Sie ein so kompliziertes Füll-Muster verwenden, daß der Amiga erst einmal ein bißchen rechnen muß.

In Version V3.0 kann der Füllvorgang durch die GFA-Break-Funktion abgebrochen werden. Die oben beschriebenen V2.xx-Probleme lassen sich also - Ostrowski macht es möglich - durch die Tastenkombination <Control><Shift><Alternate> weitest-

gehend unterbinden. Die Abbruch-Tasten müssen dabei gedrückt gehalten werden, da intern erst die aktuelle Füll-Sequenz beendet wird.

In Version V3.0, die uns für den Amiga vorliegt, kann optional mit dem Parameter "Farbe" ein Farbwert angegeben werden. Es wird dann nur diese Farbe als Füllbegrenzung gewertet. Alle anderen Punkte werden gefüllt. Liegt z.B. der Bildpunkt Xpos/Ypos auf einer Linie, wird der Füllvorgang nur auf dieser Linie durchgeführt.

Bei eingeschaltetem Clipping (siehe CLIP) wird generell nur bis an die Grenzen des CLIP-Ausschnitts gefüllt. Beispiel (nur Lowres):

```
OPENS 2,0,0,320,256,3,0
OPENW 2,0,0,320,256,0,15,2
DEFFILL 7,2,6
PBOX 10,10,310,190
DEFFILL 11,2,8
PBOX 90,40,180,90
FOR i%=1 TO 15
  DEFFILL i%,2,8
  PCIRCLE RANDOM(80)+85,RANDOM(50)+45,10
NEXT i%
DEFFILL 4,2,8
FILL 176,44,7
CLOSEW 2
CLOSES 2
```

POLYFILL { POLYF } — Polygon zeichnen, gefüllt

```
POLYFILL Pkte,Xp(),Yp() [OFFSET Xdiff,Ydiff]
```

Es gelten die gleichen Ausführungen wie zu POLYLINE, nur daß zusätzlich die zwischen den einzelnen Linien liegenden Flächen mit dem eingestellten Füllmuster ausgefüllt werden.

Ein Beispiel finden Sie unter POLYLINE.

TEXT { T } — Text im Grafikmodus ausgeben

```
TEXT Xt,Yt,"Text"
```

"Text" kann sowohl direkt als Text, als String-Variable oder als zusammengesetzter Textausdruck (A$+Str$("1")) angegeben werden. "Xt,Yt" steht für den Bildschirmpunkt, an den der Text linksbündig angelegt wird. Die Ausgaben beschränken sich auf den Bildschirm.

Der zur Ausrichtung des Textes ausschlaggebende Punkt ist immer der, der bei normaler Lage des Textes (waagerecht von links nach rechts) in der Zeichenbox des ersten Zeichens (auch bei Leerzeichen) links unten liegt. Mit diesem Punkt wird der Text generell an die angegebene X/Y-Position angelegt. Beispiel:

```
DEFFILL ,2,1
PBOX 20,5,300,120
DEFTEXT 1,16,0,16
FOR i%=1 TO 4
  GRAPHMODE i%
  tx$="TEXT (normal) GRAPHMODE "+STR$(i%)
  TEXT 50,15+i%*20,tx$
NEXT i%
```

SCROLL { SC } — BitMap-Bereich verschieben

```
SCROLL Dx,Dy,X1,Y1,X2,Y2
```

Mit Hilfe dieses Kommandos kann der Bildschirmbereich, der durch die Koordinatenpaare X1,Y1 und X2,Y2 beschrieben wird, verschoben werden. Die Verschiebe-Richtung und -Breite beschreiben die einleitenden Werte für X und Y getrennt.

Setz man für Dx oder Dy positive Werte, so bedeutet dies eine Verschiebung nach links bzw. nach unten. Umgekehrt verfahren die negativen Werte.

```
' Demonstration des Scroll-Befehls
'
' Das große GFA-BASIC Buch
'
' (p) by Wolf-Gideon Bleek im August 1989
' (c) by DATA BECKER GmbH 1989
'
COLOR 1
FOR i=1 TO 200
  DRAW  TO RANDOM(640),RANDOM(256)
NEXT i
FOR i=1 TO 600
  dx=RANDOM(11)-5
  dy=RANDOM(5)-2
  x1=RANDOM(320)
  x2=RANDOM(320)+320
  y1=RANDOM(128)
  y2=RANDOM(128)+128
  SCROLL dx,dy,x1,y1,x2,y2
NEXT i
```

Dieses Programm demonstriert nicht nur die Geschwindigkeit des Amiga beim Linienzeichen, sondern zeigt auch, welche wunderbaren Effekte sich mit einer recht einfachen Befehlskonstruktion erziehlen lassen. Bitte beachten Sie, daß die an den Seiten des verschobenen Bereichs herausgeschobenen Pixel nicht gespeichert werden. Nachgeschoben wird auf der gegenüberliegenden Seite immer die Hintergrundfarbe.

9.3 Strich-/Punktgrafik

DRAW { DR } — Punkte zeichnen und verbinden

```
DRAW TO Xpos,Ypos
DRAW X1,Y1 [TO X2`,Y2 [TO X3,Y3...]]
```

Die erste Syntax-Variante verbindet den durch Xpos/Ypos bezeichneten Punkt mit dem zuletzt durch DRAW, PLOT oder LINE gezeichneten Grafik-Punkt. Die zweite Variante zeichnet einen beliebig langen Linienzug durch die angegebene Punkte-

Kette. Wird die optionale Punkte-Kette weggelassen (DRAW X1,Y1), dann wird an der angegebenen Position ein einzelner Punkt gesetzt (vgl. PLOT).

Beispiel: Damit Sie sich nicht soviel Arbeit machen müssen, die einzelnen Punktkoordinaten einer Kette einzutippen, folgt nun ein kleines Hilfsprogramm. Wenn Sie es gestartet haben, können Sie mit einem Druck auf die linke Maustaste einzelne Punkte zeichnen. Es sind zwei Arrays dimensioniert, die die jeweilige X- und Y-Koordinate des gerade gezeichneten Punktes festhalten. Jedes dieser beiden Arrays hat 1000 Elemente, d.h. daß Sie maximal 1000 Punkte zeichnen können. Wenn Sie diese Punkteanzahl erreicht haben oder die rechte Maustaste drücken, werden die beiden Arrays in Form von DATA-Zeilen unter dem Namen DRAW.LST auf der RAM-Disk abgespeichert.

Anschließend wird das erste Programm beendet. Laden Sie nun mit Merge die geschriebene DATA-Datei in den Arbeitsspeicher und starten Sie das zweite Programm. Sie werden sehen, daß Ihre Zeichnung nun anhand der DATA-Zeilen und des DRAW-Befehls neu auf den Bildschirm gebracht wird.

Programm 1:

```
DIM px(1000),py(1000)         ! DIM Punkte-Speicher
DO                            ! Eingabe-Schleife
  MOUSE x,y,k                 ! Maus-Status holen
  IF k=1                      ! Linke Maustaste gedrückt?
    PAUSE 1                   ! Kleine Klickpause
    DRAW x,y                  ! Punkt zeichnen
    INC count                 ! Punktezähler +1
    px(count)=x               ! X-Koordinate speichern
    py(count)=y               ! Y-Koordinate speichern
  ENDIF
  EXIT IF k=2 OR count=1000 ! 1000 Punkte erreicht oder...
LOOP                          ! ...rechte Maustaste gedrückt?
OPEN "O",#1,"RAM:DRAW.LST"  ! Datei öffnen
PRINT #1;"D.rawkoos:";CHR$(13) ! DATA-Label schreiben
FOR j=1 TO 100                ! 100 DATA-Zeilen
  PRINT #1;"D ";              ! DATA schreiben
  FOR i=1 TO 10               ! 10 Koordinatenpaare je Zeile
    INC ci                    ! Indexzähler +1
    PRINT #1;STR$(px(ci));",";STR$(py(ci)); ! X/Y schreiben
    EXIT IF ci=>count         ! Abbruch, wenn Anzahl < 1000
    IF i<10                   ! Zeilenende noch nicht erreicht?
```

```
      PRINT #1;",";             ! Dann Komma schreiben
    ENDIF
  NEXT i                        ! Nächstes Koordinatenpaar
  PRINT #1;CHR$(13)             ! CR-Zeilenende schreiben
  EXIT IF ci=>count             ! Abbruch, wenn Anzahl < 1000
NEXT j                          ! Nächste DATA-Zeile
PRINT #1;"D ";STR$(1111)        ! DATA-Endmarke schreiben
CLOSE #1                        ! Datei schließen
EDIT                            ! Programmende
```

Programm 2:

```
DIM px(1000),py(1000)           ! DIM Punkte-Speicher
RESTORE d.rawkoos               ! DATA-Zeiger setzen
READ px(1),py(1)                ! 1. Koordinatenpaar lesen
PLOT px(1),py(1)                ! und zeichnen
FOR i=2 TO 1000                 ! Restliche Koordinaten
  READ px(i),py(i)              ! Lesen
  ' Pbox Px(I),Py(I),Px(I)+10,Py(I)+10
  ' Ersetzen Sie die Zeile mit dem Draw-Befehl durch diese
  ' Pbox-Zeile, und schon haben Sie einen neuen Effekt.
  DRAW  TO px(i),py(i)          ! Und zeichnen
  EXIT IF px(i)=1111            ! Abbruch, wenn Ende erreicht
NEXT i                          ! Nächstes Paar
```

DRAW $ { DR } — Plotter-(Turtle-)Grafik

`DRAW Def$[,Const[,"Def"[,Var[,...]]]]` *Version 3.0*

Erlaubt eine Plotter-Simulation auf dem Bildschirm (LOGO-Turtle-Grafik). In Def$/"Def" können wahlweise als alphanumerischer Ausdruck, als String-Variable oder Textkonstante Turtle-Kommandos angegeben werden. Als Kommandos sind folgende Kürzel vorgesehen:

fd n (forward) Bewege Stift n Pixel vorwärts.

bk n (backward) Bewege Stift n Pixel rückwärts.

sx n Skalierung aller bei fd und bk angegeben Werte in X-Richtung mit dem Wert n.

sy n Skalierung aller bei fd und bk angegeben Werte in Y-Richtung mit dem Wert n.

sx 0 X-Skalierung ausschalten.

sy 0 Y-Skalierung ausschalten.

lt n (left turn) Drehe Stift um n Grad nach links.

rt n (right turn) Drehe Stift um n Grad nach rechts.

tt n (turn to) Setze Stift absolut in Richtung n Grad (Gradeinteilung siehe SETDRAW).

ma x,y (move absolute) Bewege Stift (pu) auf absolute Position x/y (siehe auch SETDRAW).

da x,y (draw absolute) Bewege Stift (pd) auf absolute Position x/y, und zeichne dabei eine Linie.

mr x,y (move relative) Bewege Stift (pu) auf Position x/y, relativ zur aktuellen Turtle-Position.

dr x,y (draw relative) Bewege Stift (pd) auf Position x/y, relativ zur aktuellen Turtle-Position, und zeichne dabei eine Linie.

co n (color) Linienfarbe n einstellen (siehe COLOR).

pu (pen up) Stift anheben (Stift schwebt).

pd (pen down) Stift aufsetzen.

(n enthält den jeweils anzugebenden Wert, x,y die betreffenden Koordinaten).

Eine DRAW-Zeile könnte folgendermaßen zusammengesetzt sein:

```
Draw "ma",X%,Y%,"tt",Int(10.52),"pd",A$,Len(B$),"lt60 fd3.4"
```

Die Angabe der Entfernungen, Winkel und Koordinaten etc. kann ebenfalls wahlweise als Ausdruck, als Konstante, als Variable oder innerhalb von Def$/"Def" erfolgen. Dabei ist die Anzahl der durch Komma getrennten Einzelanweisungen beliebig (max. Eingabezeilenlänge = 255 Zeichen). Das Komma kann in den meisten Fällen auch vernachlässigt werden. Beispiel:

```
GRAPHMODE 3                                ! XOR-Modus
FOR i%=0 TO 640 STEP 5                     ! Einmal von links nach rechts
  FOR j%=0 TO 1                            ! Zweimal hintereinander
    SETDRAW 100+i%,170+COS(i%*PI/180)*50,i%+90 ! Turtle setzen
    DRAW "pd rt90 fd20 rt90 fd30 lt45 fd14.1 lt45 fd40"
    DRAW "lt45 fd14.1 lt45 fd10 lt90 fd20 lt90 fd10 rt90"
    DRAW "fd20 rt90 fd30 rt90 fd60 rt90 fd40 rt45 fd28.3"
    DRAW "rt45 fd60 rt45 fd28.3 rt45 fd40"
    DRAW "pu fd30 pd fd40 rt90 fd20 rt90 fd30 lt45 fd14.1"
    DRAW "lt45 fd10 lt90 fd20 rt90 fd20 rt90 fd20 lt90 fd40"
    DRAW "rt90 fd20 rt90 fd80 rt45 fd28.3 rt45 fd40"
    DRAW "pu fd30 pd fd30 rt45 fd28.3 rt45 fd80 rt90 fd20"
    DRAW "rt90 fd40 lt90 fd30 lt90 fd40 rt90 fd20 rt90 fd80"
    DRAW "rt45 fd28.3 pu rt135 fd30 pd fd10 lt90 fd30 lt90"
    DRAW "fd10 lt45 fd14.1 lt45 fd10 lt45 fd14.1 pu"
  NEXT j%
NEXT i%
```

DRAW() — Plotter-(Turtle-)Attribute liefern

`Var=DRAW(Index)` *Version 3.0*

Liefert Informationen über die aktuellen DRAW-Turtle-Attribute.

Index:

- 0 = Aktuelle X-Position
- 1 = Aktuelle Y-Position
- 2 = Aktuell eingestellter Winkel (in Grad)
- 3 = Aktuelle X-Skalierung
- 4 = Aktuelle Y-Skalierung
- 5 = Pen-Flag (-1 = pd/0 = pu)

Mit Index 0 bis 4 werden Fließkommawerte geliefert. Beispiel: Aus der Sprache LOGO kennen Sie vielleicht die Turtle (Schildkröte), die in den meisten Fällen als Pfeil dargestellt wird. Mit den folgenden beiden Prozeduren können Sie diese optische Turtle simulieren. Die Prozedur Showt speichert den aktuellen Hintergrund unter der Turtle und zeichnet einen Pfeil auf die aktuelle Position. Mit Hidet wird der Hintergrund wieder restauriert und die Turtle damit gelöscht.

```
SETDRAW 20,20,90                   ! Turtle setzen
PRINT "TURTLE-Steuerung per Maustasten"
showt                              ! Turtle zeigen
DO
  dxmax=MAX(15,MIN(624,DRAW(0))) ! Immer schön..
  dymax=MAX(15,MIN(240,DRAW(1))) ! ...im Bild...
  SETDRAW dxmax,dymax,DRAW(2)    ! ...bleiben
  hidet                          ! Turtle verstecken
  DRAW "fd2"                     ! 2 Pixel zeichnen
  showt                          ! Turtle zeigen
  IF MOUSEK=1                    ! Linke Maustaste?
    DRAW "lt2"                   ! Dann 2 Grad linksrum
  ELSE IF MOUSEK=2               ! Rechte Maustaste?
    DRAW "rt2"                   ! Dann 2 Grad rechtsrum
  ENDIF
LOOP
'
PROCEDURE showt                    ! TURTLE-Proc
  LOCAL xtrtl,ytrtl,xtrtl2,ytrtl2,wtrtl,ptrtl
  xtrtl=DRAW(0)                    ! Aktuelle X-Position?
  ytrtl=DRAW(1)                    ! Aktuelle Y-Position?
  xtrtl1=MAX(0,MIN(639,xtrtl-15)) !——. GET-
  ytrtl1=MAX(0,MIN(255,ytrtl-15))     ! Bereich..
  xtrtl2=MAX(0,MIN(639,xtrtl1+30))    ! eingrenzen
  ytrtl2=MAX(0,MIN(255,ytrtl1+30))!——'
  wtrtl=DRAW(2)                    ! Aktueller Winkel?
  ptrtl%=DRAW(5)                   ! Aktueller Pen-Status
  GET xtrtl1,ytrtl1,xtrtl2,ytrtl2,trtlbckgrnd$
  '                                ! Hintergrund speichern
  DRAW "pd lt120 fd11 rt150 fd20 rt120 fd20 rt150 fd11"
  '                                ! Turtle malen
  SETDRAW xtrtl,ytrtl,wtrtl        ! Alten Turtle-Status setzen
  IF ptrtl%=0                      ! Stift war vorher oben?
    DRAW "pu"                      ! Dann wieder hochsetzen
  ENDIF
RETURN
'
PROCEDURE hidet                 ! Turtle löschen
  PUT xtrtl1,ytrtl1,trtlbckgrnd$
RETURN
```

LINE { LI } — Linie zeichnen

```
LINE X1,Y1,X2,X2
```

Die Bildschirm-Koordinaten X1,Y1 und X2,Y2 werden durch eine gerade Linie im aktuellen - zuletzt mit DEFLINE - eingestellten Linienstatus gezeichnet. Beispiel:

```
FOR j%=1 TO 20
  LINE 10,j%*10,100,j%*10
  LINE j%*10,10,j%*10,100
NEXT j%
```

PLOT { PL } — Punkt zeichnen

```
PLOT Xpos,Ypos
```

Xpos/Ypos bestimmt die Lage eines Bildschirm-Punktes, der gesetzt werden soll.

POINT() — Bildschirmpunkt-Farbwert ermitteln

```
Var=POINT(Xpos,Ypos)
```

Liefert die Nummer des Farbregisters, aus dem der durch Xpos/Ypos bezeichnete Bildschirm-Punkt seine Farbe bezieht (siehe COLOR). Beispiel:

```
t%=TIMER
DEFFILL ,2,2            ! DEFFILL hellgrau
PBOX 10,10,100,100      ! Kleine PBOX
FOR x%=10 TO 100        ! Alle Punkte in X-Richtung
  FOR y%=10 TO 100      ! Alle Punkte in Y-Richtung
    IF POINT(x%,y%)=0 ! Testen, ob nicht gesetzt
      PLOT 110+x%,y%  ! Ja, dann setzen
    ENDIF
```

```
    NEXT y%              ! Nächste Spalte
  NEXT x%                ! Nächste Zeile
  PRINT (TIMER-t%)/200'"Sek."
```

Das Ergebnis des Programms ist das Negativ der zuerst gezeichneten PBOX, da statt jedes Punktes, der in der PBOX weiß ist, in der neuen Box ein Punkt gesetzt wird.

POLYLINE { POL } — Polygon zeichnen

```
POLYLINE Pkte,Xp(),Yp() [OFFSET Xdiff,Ydiff]
```

Die POLY-Befehle (POLYFILL, POLYLINE) verwenden jeweils die Füll-, Linien- oder Marker-Attribute, die vorher mit DEFLINE, DEFFILL oder DEFMARK vorgenommen wurden.

In den Integer-Feldern Xp() und Yp() sind der Reihe nach so viele Koordinatenpaare anzugeben, wie Polygon-Ecken vorgesehen sind. Der erste zu definierende Punkt des Vielecks ist im Element 0 (OPTION BASE 0) bzw. im Element 1 (OPTION BASE 1) der Felder abzulegen, da die Inhalte der ersten beiden Feldelemente als Koordinatenwerte des ersten Punktes verwendet werden.

Die Anzahl der zu berücksichtigenden Polygon-Eckpunkte wird in Pkte übergeben. Maximal dürfen 128 Punkte angegeben werden. Soll mit POLYLINE ein geschlossener Linienzug gezeichnet werden, muß der letzte Punkt mit dem ersten identisch sein (Xp(n)=Xp(0)/Yp(n)=Yp(0)).

Um sich nicht jedesmal die Arbeit machen zu müssen, die Koordinatenpaare neu zu bestimmen, sobald dieselbe Figur an einer anderen Stelle gezeichnet werden soll, kann mit dem Zusatzbefehl OFFSET, der ggf. den POLY-Befehlen angehängt wird, in X- und in Y-Richtung ein Versatz der Figur um einen bestimmten Betrag (negativ oder positiv) bewirkt werden. Beispiel:

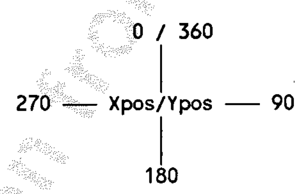
```
DIM x(10),y(10)                       ! DIM Punkt-Felder
GRAPHMODE 3                           ! XOR-Modus
DO                                    ! Endlos-Schleife
  FOR i=0 TO 9                        ! 10 Punkte
    x(i)=RANDOM(100)+56*i             ! Zufällige X-Koordinate
    y(i)=RANDOM(128)+40               ! Zufällige Y-Koordinate
  NEXT i
  x(i)=x(0)                           ! Letztes X = 1. X
  y(i)=y(0)                           ! Letztes Y = 1. Y
  bb=INT(RND*24)                      ! Zufälliges Füllmuster
  cc=RANDOM(5)+1                      ! Zufälliger Linientyp
  DEFFILL 1,2,bb                      ! Muster-DEF
  DEFLINE cc                          ! Linien-DEF
  POLYFILL 9,x(),y()                  ! Fläche zeichnen
  FOR j%=1 TO 2                       ! 2 mal
    FOR i%=3 TO 99 STEP 6             ! 16 Offsets
      POLYLINE 11,x(),y() OFFSET i%,i% ! Vieleck zeichnen
    NEXT i%                           ! Nächstes Offset
  NEXT j%
  POLYFILL 9,x(),y()                  ! Fläche löschen
LOOP
```

SETDRAW { SETD } — DRAW-Turtle positionieren

`SETDRAW Xpos,Ypos,Grad` *Version 3.0*

Setzt den Stift für Plotter-(Turtle-)Grafik auf die absolute Position Xpos,Ypos in die Zeichenrichtung Grad (siehe auch DRAW $).

Grad:



Ein Beispiel hierzu finden Sie unter DRAW().

9.4 Grafikoperationen

CLIP { CLI } — **Grafikausgabe begrenzen/Nullpunkt setzen**

Version 3.0

```
CLIP Xpos,Ypos,Breite,Höhe [OFFSET X,Y]
CLIP Xl,Yo TO Xr,Yu [OFFSET X,Y]
CLIP #Windownummer [OFFSET X,Y]
CLIP OFFSET X,Y
CLIP OFF
```

Ermöglicht die Bestimmung eines Bildschirmrechtecks, auf das dann sämtliche Grafikausgaben begrenzt werden (außer PUT) bzw. die Festlegung des Koordinaten-Nullpunktes für Grafikausgaben. Alle Teile von Grafikausgaben (LINE, BOX, PBOX, DRAW etc.), die die Grenzen dieses Rechtecks überschreiten, werden an dessen Grenzen abgeschnitten.

Syntax-Variante 1: Xpos und Ypos beschreiben die Position sowie Breite und Höhe des CLIP-Rechtecks.

Syntax-Variante 2: Xl und Yo beschreiben die Position der linken oberen Ecke des CLIP-Rechtecks. Xr und Yu (hinter TO) beschreiben die Position seiner rechten unteren Ecke.

Syntax-Variante 3: #Windownummer enthält die GFA-Fensternummer, dessen Arbeitsbereichs-Position und -Ausmaße für das CLIP-Rechteck gelten sollen.

Syntax-Variante 4: X und Y legen den Koordinaten-Ursprung (in Relation zur linken oberen Bildschirmecke) für künftige Grafikausgaben fest (außer PUT). OFFSET X,Y kann auch optional an die vorangegan-

genen CLIP-Befehlsvarianten angehängt werden, wodurch ebenfalls der Grafik-Nullpunkt bestimmt wird.

Syntax-Variante 5: CLIP OFF schaltet aktuelles Clipping wieder aus. Beispiel:

```
DEFFILL ,2,4              ! DEFFILL grau
PBOX 2,2,218,118          ! Hintergrund-Fläche zeichnen
DEFFILL ,2,2              ! DEFFILL hellgrau
CLIP 10,10,100,100        ! Ab 10/10 100 Pixel in jeder
'                         ! Richtung als Clip-Box
PCIRCLE 60,60,60          ! Gefüllten Kreis zeichnen
CLIP 110,10 TO 210,110 OFFSET 100,10
' Bildschirmbereich mit den Koordinaten 110/10 und 210/110
' als Clip-Bereich anmelden und Nullpunkt auf 100/10 setzen.
PCIRCLE 60,50,60          ! Gefüllten Kreis zeichnen
CLIP OFF                  ! Clipping ausschalten
CLIP  OFFSET 0,0          ! Grafik-Nullpunkt wieder auf 0/0 setzen
BOX 10,10,210,110         ! Box um die beiden Kreis-Ausschnitte  '
```

GET — Bildschirmbereich speichern

```
GET X_links,Y_oben,X_rechts,Y_unten,Var$
```

Durch GET (nicht zu verwechseln mit dem Diskettenbefehl GET#) wird mit den Koordinatenangaben X_links/Y_oben und X_rechts/Y_unten ein Bildschirmausschnitt definiert, der als Bitmuster in Var$ eingelesen wird. Die angegebenen Eck-Koordinaten müssen innerhalb des Bildschirmbereichs liegen, da sonst der Interpreter den Befehl nicht ausführt.

Diese sparsame Beschreibung läßt nicht ahnen, was mit diesem Befehl alles machbar ist. Dies ist für die Grafik-Programmierung einer der wichtigsten Befehle überhaupt.

Beispiel: Und wieder mal ein "kleines" Demo-Listing. Es produziert eines der so heißbegehrten Pop-Up-Menüs. Dies sind Menüs, die nicht - wie vom Intuition her gewohnt - vom oberen Bildrand "herunterfallen" bzw. "heruntergezogen" werden (Drop-

Down-Menü, bzw. Pull-Down-Menü), sondern an der Bildschirmposition erscheinen, an der sie benötigt werden, nämlich unter dem Mauszeiger.

Dabei kann ein beliebiger Bildschirmbereich angegeben werden, innerhalb dessen das Menü dargestellt werden soll bzw. der als Reaktionsbereich fungiert. Ist der Mauszeiger zum Zeitpunkt des Aufrufs innerhalb dieses Bereichs, so bleibt das Menü unter dem Mauszeiger stehen. Wird ein Menüpunkt angewählt oder verläßt der Mauszeiger den Bereich wieder (auch ohne Auswahl), so wird das Menü wieder gelöscht. Unter welchen Bedingungen Sie den Menü-Aufruf zulassen, bleibt Ihnen überlassen.

Bei der Einrichtung des Menüs habe ich versucht, weitestgehend den Konventionen der GFA-PullDown-Menü-Definition Rechnung zu tragen. So wird z.B. auch hier ein Menüzeilentext, der mit einem Bindestrich beginnt, als unwählbar (grau) dargestellt. Der Punkt ist dann von der Auswahl ausgeschlossen.

Ein wesentlicher Unterschied ist, daß hier aus einem DATA-Block, der die einzelnen Zeilentexte enthält, über einen wählbaren Startindex beliebige Menüteile ausgeschnitten und angezeigt werden können. Außerdem ist die Anzeige des Menüzeilentextes auf eine Pixel-Breite von 100 Pixel (Hires/Midres) bzw. 50 Pixel (Lowres) beschränkt. Der Text wird per TEXT auf diese Breite formatiert und in den Menüpunkt eingesetzt.

```
start%=1                   ! Erst ab 2. Menüpunkt darstellen
anzahl%=20                 ! Insgesamt 6 Menüpunkte darstellen
DIM feld$(start%+anzahl%)  ! Textfeld einrichten. Nur so viele
'                   Menüpunkt-Elemente, wie zur Demo nötig sind.
RESTORE m.datas            ! Datazeiger setzen
FOR i%=1 TO start%+anzahl% ! Menütext-Datas
  READ feld$(i%)           ! Einlesen
  PRINT feld$(i%)
NEXT i%
m.datas:
DATA ---,Speichern,Laden,Löschen,Kopieren,----------,QUIT
DATA ---,Speichern,Laden,Löschen,Kopieren,----------,QUIT
DATA ---,Speichern,Laden,Löschen,Kopieren,----------,QUIT
```

Der Einfachheit halber habe ich hier dreimal dieselben Texte verwendet, um den folgenden Block der IF..ENDIF-Abfragen in Grenzen zu halten. In V3.0 können diese Abfragen auch leicht durch IF..ELSE IF oder SELECT..CASE vereinfacht werden.

Im folgenden Listing mußten wieder einige Programmzeilen aus drucktechnischen Gründen getrennt werden. Die drei Trennpunkte am Zeilenende und Zeilenbeginn gehören nicht zur Zeilensyntax.

```
BOX 50,50,300,200
DO
  REPEAT
    IF MOUSEX>50 AND MOUSEX<300 AND MOUSEY>50 AND MOUSEY<399
      @menue(start%,anzahl%,50,50,300,255,*feld$(),*index%)
    ENDIF
    '  Weitere Aufruf-Variante:
    '  If Mousek=2
    '    @Menue(Start%,Anzahl%,0,0,639,255, *Feld$(),*Index%)
    '  Endif
  UNTIL index%>0              ! Or Mousek
  IF feld$(index%+start%)="Speichern"
    feld$(index%+start%)="-Speichern"
    feld$(index%+start%+1)="Laden"
    PRINT AT(1,1);"gewählt: Speichern     "
  ENDIF
  IF feld$(index%+start%)="Laden"
    feld$(index%+start%-1)="Speichern"
    feld$(index%+start%)="-Laden"
    PRINT AT(1,1);"gewählt: Laden         "
  ENDIF
  IF feld$(index%+start%)="Löschen"
    PRINT AT(1,1);"gewählt: Löschen       "
  ENDIF
  IF feld$(index%+start%)="Kopieren"
    feld$(index%+start%)=CHR$(8)+" Kopieren"
    PRINT AT(1,1);"gewählt: Kopieren      "
    GOTO label
  ENDIF
  IF feld$(index%+start%)=CHR$(8)+" Kopieren"
    feld$(index%+start%)="Kopieren"
    PRINT AT(1,1);"gewählt: Kopieren      "
  ENDIF
  IF feld$(index%+start%)="QUIT"
    PRINT AT(1,1);"gewählt: Quit          "
  ENDIF
  label:
  PRINT "Menüindex : ";index%
  EXIT IF feld$(index%+start%)="QUIT"
```

```
  CLR index%
LOOP
'
PROCEDURE menue(pm1%,mmx%,mxl%,myo%,mxr%,myu%,f.adr%,v.adr%)
  ' Pop-Up-Menü
  '
  ' Pm1%   = Index des Menüpunktes (Textfeldindex), der an
  '          erster Stelle erscheinen soll. Die Texte müssen
  '          ab Index 0 im Textfeld stehen.
  ' Mmx%   = Anzahl der Menüpunkte, die ab Pm1% dargestellt
  '          werden sollen. Der Menü-Index liegt immer im
  '          Bereich von 1 bis Mmx%.
  ' Mxl%, Myo%, Mxr%, Myu%
  '        = Rahmenkoordinaten, innerhalb derer das Menü
  '          dargestellt werden soll. Paßt das Menü nicht
  '          in den Rahmen, liegt die rechte untere Ecke
  '          des Menüs immer auf der rechten unteren Ecke
  '          des Rahmens.
  ' F.adr% = Pointer auf das Textfeld.
  ' V.adr% = Pointer auf eine Rückgabevariable.
  '          Die Rückgabevariable enthält nach Abschluß den
  '          Index des gewählten Menüpunktes. Um den dazugehörigen
  '          Textfeld-Index zu ermitteln, muß zum Menüindex 'Pm1%'
  '          addiert werden.
  LOCAL mmen$,msk%,m.key$,yi%,yi2%,mrs%,m.i%,lsr%
  LOCAL mxl2%,mxr2%,myo2%,myu2%
  DIM dum$(1)                  ! Lokales Swap-Feld
  SWAP *f.adr%,dum$()          ! Felder swappen
  mxl2%=MIN(MAX(MOUSEX-68,mxl%),mxr%-136)
  mxr2%=mxl2%+136
  myo2%=MIN(MAX(MOUSEY-6,myo%),myu%-18+mmx%*18)
  myu2%=myo2%+(18+mmx%*18)
  mxl%=MIN(mxl%,mxl2%)
  myo%=MIN(myo%,myo2%)
  '
  ' Die letzten 6 Zeilen haben die schwierige Aufgabe, den
  ' gewünschten Darstellungsbereich des Menüs und die Position
  ' der Maus so miteinander zu verknüpfen, daß das Menü -
  ' wenn möglich - unter dem Mauszeiger, aber nicht außerhalb
  ' des gewählten Rahmens dargestellt wird.
  '
  GET MAX(0,mxl2%),MAX(0,myo2%),MIN(639,mxr2%),MIN(myu2%,255),mmen$
  ' Menü-Hintergrund speichern
  DEFFILL 1,0,0                 ! DEFFILL weiß
  GRAPHMODE 1                   ! Replace-Modus
  PBOX mxl2%,myo2%,mxr2%,myu2% !------------------.
  BOX mxl2%+1,myo2%+1,mxr2%-1,myu2%-1             |
  DEFFILL 1,2,4                 ! DEFFILL grau    |
  PBOX mxl2%+6,myo2%+6,mxr2%-6,myu2%-6            |
  DEFFILL 1,0,0                 ! DEFFILL weiß    |
  FOR m.i%=1 TO mmx%            ! Alle Menü-Zeilen|
    GRAPHMODE 1                 ! Replace-Modus   |
    PBOX mxl2%+13,myo2%-6+m.i%*18,mxr2%-13,myo2%+6+m.i%*18
```

```
    GRAPHMODE 2                    ! Transparent-Modus        |
    IF LEFT$(dum$(m.i%+pm1%))="-" ! 1.Zeichen =  -  ?  |- Menü
      TEXT
mxl2%+20,myo2%+3+m.i%*18,RIGHT$(dum$(m.i%+pm1%),LEN(dum$(m.i%+pm1%))-1)
    ELSE                           ! Aktive Zeile !           |
      TEXT mxl2%+20,myo2%+3+m.i%*18-1,dum$(m.i%+pm1%)        |
    ENDIF                                                      |
  NEXT m.i%                        ! Nächste Zeile ----------'
  DEFMOUSE 3                       ! DEFMOUSE Zeigefinger
  DEFFILL 1,1,1                    ! DEFFILL schwarz
  GRAPHMODE 3                      ! XOR-Modus
  BOUNDARY 0                       ! V3.0 : P-Rahmen aus
  REPEAT                           ! Auswahl-Schleife >-----------.
    ON MENU                        ! Ereignis-Überwachung         |
    MOUSE xko.x%,yko.y%,msk%       ! Maus-Status holen            |
    yi%=INT((yko.y%-(myo2%-8))/18)                                |
    '                              ! Zeilen-Index berechnen       |
    IF xko.x%>mxl2%+12 AND yi%>0 AND xko.x%<mxr2%-13 AND yi%<=(mmx%) |
      ' Maus auf einem Menüpunkt?  |
      IF LEFT$(dum$(yi%+pm1%))<>"-"  ! Menüpunkt aktiv?           |
        PBOX mxl2%+14,myo2%-5+yi%*18-1,mxr2%-14+1,myo2%+5+yi%*18
      ENDIF
      REPEAT                       ! Mausbewegung abwarten >---.  |
        yi2%=INT((MOUSEY-(myo2%-8))/18)                        |  |
        '                          ! Ggfs. neuen Index holen   |  |
        msk%=MOUSEK                ! Maustasten-Status holen   |  |
        m.key$=INKEY$              ! Tastatur abfragen         |  |
        ON MENU                    ! Ereignis-Überwachung      |  |
      UNTIL MOUSEX<mxl2%+12 OR MOUSEX>mxr2%-13 OR yi%<>yi2% OR msk%>0 OR
m.key$>"" !<-' |
      IF LEFT$(dum$(yi%+pm1%))<>"-"  ! Menüpunkt aktiv?           |
        PBOX mxl2%+14,myo2%-5+yi%*18-1,mxr2%-14+1,myo2%+5+yi%*18
      ELSE                         ! Menüpunkt ist inaktiv!       |
        CLR msk%                   ! Maustasten-Status löschen    |
      ENDIF                                                       |
    ELSE                           ! Maus nicht auf Menüpunkt!    |
      CLR yi2%                     ! Punkt-Index löschen          |
    ENDIF                                                         |
    EXIT IF (MOUSEX<mxl% OR MOUSEX>mxr% OR MOUSEY<myo% OR MOUSEY>myu%)
AND yi2%<=mmx%
    ' Schleife verlassen, wenn sich der Mauszeiger außerhalb      |
    ' des Darstellungsbereichs befindet.                          |
    m.key$=INKEY$                  ! Tastatur abfragen            |
  UNTIL (msk%>0 OR m.key$>"") AND yi2%<=mmx%                      |
  '   Schleife verlassen, wenn Tastatur betätigt oder ein         |
  '   Mausknopf gedrückt wurde.! <--------------------------------'
  DEFMOUSE 0                       ! DEFMOUSE Pfeil
  DEFFILL 1,0,0                    ! DEFFILL weiß
  GRAPHMODE 1                      ! Replace-Modus
  PUT MAX(0,mxl2%),MAX(0,myo2%),mmen$ ! Hintergrund restaurieren
  BOUNDARY 1                       ! V3.0 : P-Rahmen an
  SWAP *f.adr%,dum$()              ! Menütextfeld wieder global
  ERASE dum$()                     ! Lokales Swap-Feld löschen
```

```
  *v.adr%=yi2%                    ! Gewählten Index zurückgeben
  PAUSE 5                         ! Kleine Klickpause
RETURN
```

Weitere Beispiele zu GET finden Sie hier im Buch in Hülle und Fülle.

PUT { PU } — Bildschirmbereich setzen

```
PUT X_links,Y_oben,Var$ [,Modus,Maske]
```

PUT (nicht zu verwechseln mit dem Diskettenbefehl PUT#) zeichnet einen durch GET (siehe dort) eingelesenen Bildausschnitt an die Koordinaten X_links/Y_oben. Die bei GET definierte Größe bleibt dabei unverändert. Die Koordinaten des unteren rechten Eckpunktes ergeben sich aus der Breite und Höhe des mit GET gespeicherten Ausschnitts. Dabei kann die Position auch so gewählt werden, daß der Bildausschnitt teilweise oder auch völlig außerhalb des Bildschirmbereichs liegt.

Durch die Option Modus kann ein Grafikmodus bestimmt werden, sonst wird im Replace-Modus gezeichnet. Negative Werte für Modus sollten tunlichst vermieden werden, da sonst mit Fehlfunktionen zu rechnen ist.

Modus:

DestInvert (&H30) — In das Bild wird der invertierte Bildschirmbereich gesetzt.

SourceInvert (&H50) — Das Bild wird im Bereich des Ausschnittes invertiert.

ExclDestSource (&H60) — Der Bildschirmbereich wird mit dem Ausschnitt exklusiv-oder verknüpft.

OrDestSource (&H80) Die Verknüpfung erfolgt nur zwischen Bits, die bei Bildschirm und Ausschnitt gleich sind.

DestSource (&HC0) Der Ausschnitt wird über den Bildschirmbereich gesetzt.

9.4.1 Organisation eines PUT-Strings

In diesem Zusammenhang ist es vielleicht angebracht, ein paar Worte zur PUT-String-Organisation zu verlieren. Die ersten 3 Words (6 Bytes) des Strings enthalten der Reihe nach:

Word 1 Die Breite des Ausschnitts minus 1 (= X_rechts minus X_links).

Word 2 Die Höhe des Ausschnitts minus 1 (= Y_unten minus Y_oben).

Word 3 Anzahl der BitPlanes.

Nur bei einer BitPlane entspricht ein gesetztes Bit im PUT-Raster einem gesetzten Punkt im gespeicherten Bild-Ausschnitt. In allen anderen Fällen bildet die Kombination von übereinander liegenden Bits die Nummer des Farbregisters, aus dem die Farbe für diesen Punkt geholt werden soll.

An diesen sogenannten "Header" (Kopf) werden der Reihe nach die Bit-Informationen des gespeicherten Ausschnitts angehängt.

Dies geschieht bei einer BitPlane - das ist am einfachsten zu erklären - nach folgendem Schema: Angenommen, der Ausschnitt ist 26 Pixel breit und 15 Pixel hoch. Die Breite des Auschnitts wird durch 16 (Wordbreite) geteilt und - falls bei der Teilung ein Rest geblieben ist - zum integrierten Teilungsergebnis der Wert 1 hinzugezählt:

```
Word 1  =  Int(26/16)+Abs((26 MOD 16)>0)
```

Die Bit-Raster-Breite im PUT-String ist also immer glatt durch 16 teilbar. Im obigen Beispiel belegen die Pixel-Bits 17-26 der ersten Ausschnittzeile die Bits 0 bis 9 des 5. String-Words, während die verbleibenden Bits 10 bis 15 des 5. Words mit "irgend etwas" gefüllt werden. Diese nicht benötigten Bits enthalten also unbrauchbaren Bit-Müll.

```
|<-------------------- 32 Bit  ( = 2 Words ) ------------------->|
|<------- Ausschnittbreite = 27 Pixel --------->|<- 'Müll'->|
|                                                |           |
* * * * * * * * * * * * * * * * |* * * * * * * * * * * 0 0 0 1 0 0   1
* * * * * * * * * * * * * * * * |* * * * * * * * * * * 0 1 0 0 0 0   2
* * * * * * * * * * * * * * * * |* * * * * * * * * * * 0 0 0 1 0 0   3
* * * * * * * * * * * * * * * * |* * * * * * * * * * * 0 0 0 0 1 0   4
* * * * * * * * * * * * * * * * |* * * * * * * * * * * 0 0 1 0 0 0   5
* * * * * * * * * * * * * * * * |* * * * * * * * * * * 0 0 1 1 0 0   6
* * * * * * * * * * * * * * * * |* * * * * * * * * * * 0 0 0 0 0 0   7
* * * * * * * *   B I L D - D A T E N   * * * * * * 0 0 0 0 0 0   8
* * * * * * * * * * * * * * * * |* * * * * * * * * * * 0 1 0 1 0 0   9
* * * * * * * * * * * * * * * * |* * * * * * * * * * * 0 0 0 0 0 0  10
* * * * * * * * * * * * * * * * |* * * * * * * * * * * 1 0 0 0 1 0  11
* * * * * * * * * * * * * * * * |* * * * * * * * * * * 1 0 0 0 0 0  12
* * * * * * * * * * * * * * * * |* * * * * * * * * * * 1 0 1 0 1 1  13
* * * * * * * * * * * * * * * * |* * * * * * * * * * * 0 1 1 1 0 0  14
* * * * * * * * * * * * * * * * |* * * * * * * * * * * 1 1 0 0 1 0  15
|<--vorderes Word der Zeile--->|<--hinteres Word der Zeile--->|
0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 2 3 4 5 6 7 8 9 A B C D E F
(Hex-Bitnummern)
```

In diesem Rhythmus werden nun alle Bildzeilen im Ziel-String abgelegt. Bit 0 - 15 des 6. Words und Bit 0 - 9 des 7. Words enthalten die Daten der 2. Pixel-Zeile, Bit 0 - 15 des 8. Words und Bit 0 - 9 des 9. Words enthalten die Daten der 3. Pixel-Zeile usw.

Bei mehr BitPlanes wird das Verfahren wesentlich komplizierter. Ich habe fast einen ganzen Tag damit verbracht, mir zu überlegen, wie ich dieses Verfahren transparent machen könnte - und zwar so, daß es für jeden leicht verständlich ist. Nach vielen erfolglosen Ansätzen habe ich mich entschlossen, die vielen Seiten Text, die ich dabei verschwenden würde, durch eine möglichst aussagekräftige Grafik zu ersetzen und diese dann zu erklären.

Prinzipiell ist es in Farbe das gleiche Schema, nur daß sich die Farbinformationen eines Bildschirmpunktes daraus ergeben, daß 2 oder mehrere - wie in der Grafik dargestellt - 4-Bitraster (Planes) im oben beschriebenen Format "übereinander" gelegt werden. Das für den jeweiligen Punkt zuständige Farbregister ergibt sich aus der Bit-Summe der jeweiligen Plane-Bits. Grundsätzlich ist ein Farb-GET/PUT-Image genauso organisiert wie der Farb-Bildschirm, wobei nur jeweils die nicht benötigten Rest-Bits (Bit-Müll) des letzten Words einer Zeile zu beachten sind.

In der Grafik habe ich die Bits der Reihenfolge nach durchnumeriert. Als Beispiel habe ich ein 14*6-Raster gewählt. Nehmen Sie an, Sie "schneiden" mittels GET aus dem Bildschirm ein Teil mit der Breite von 14 und der Höhe von 6 Pixel aus. Die ersten sechs Bytes des String-Headers habe ich ja oben schon erklärt. Daran schließen sich nun - im Beispiel - im 4-Word-Rhythmus die Bit-Informationen der einzelnen Zeilen an.

Legt man nun die vier Planes direkt übereinander, so ergibt die Bit-Stellung der vier zusammengehörigen Bits die Nummer des Farbregisters, aus dem dieser Bildschirmpunkt seine Farbe bezieht. In der Grafik habe ich das Ausschnitt-Pixel 4/3 besonders herausgestellt, um daran die Farbberechnung zu demonstrieren.

```
In Plane 1 ist dieses Bit gesetzt   :  2^0  =  1
In Plane 2 ist es ebenfalls gesetzt :  2^1  =  2
In Plane 3 ist es nicht gesetzt     :    0  =  0
In Plane 4 ist es wiederum gesetzt  :  2^3  =  8
                                      ___________
Die Summe dieser vier Bits ergibt   :         11
                                      ===========
```

Das Pixel 4/3 des Ausschnitts bezieht seine Farbe also aus dem Farbregister 11. Zur Vertiefung nehme ich nun noch die vier Bits 205, 221, 237 und 317 am rechten Rand.

```
In Plane 1 ist das Bit nicht gesetzt :    0  =  0
In Plane 2 ist es gesetzt            :  2^1  =  2
In Plane 3 ist es ebenfalls gesetzt  :  2^2  =  4
In Plane 4 ist es nicht gesetzt      :    0  =  0
                                       -----------
Für Pixel 14/4 ist das Farbregister 6 zuständig.
                                       ===========
```

In der Grafik sehen Sie, daß die beiden letzten Bits jeder Zeile leer sind. An Speicherplatz wird jedoch für jedes "angebrochene" Word ein volles Word benötigt. Wäre also in diesem Fall der Ausschnitt 17 Punkte breit, so würde dafür der doppelte Speicherplatz erforderlich, obwohl vom zweiten Word jeder Zeile nur das 1 Bit belegt wäre.

Anhand der Word-Numerierung können Sie erkennen, in welcher Folge die Words im Speicher liegen.

Dieses Grundwissen soll Ihnen genügen, um die internen Verborgenheiten der Amiga-Grafik halbwegs zu verstehen. Die Befehle funktionieren aber auch, wenn Sie es nicht verstanden haben, denn ein komplexer Befehl ist ja gerade dafür da, daß er einem aufwendige Arbeit und großes Hintergrundwissen abnimmt.

9.4.2 Organisation des Bildschirm-Speichers

SPRITE { SPR }	**Sprite setzen und löschen**

```
SPRITE #Nummer, Def.var$
SPRITE #Nummer [, ""]
SPRITE #Nummer [,Xpos,Ypos]
SPRITE ON/OFF
```

Mit Def.var$ wird eine String-Variable angegeben, deren Inhalt im MKI$-Format die Mausform definiert.

Grundsätzlich sind Sprite(s) nichts anderes als der Mauszeiger, der ja ebenfalls den Charakter eines Sprites besitzt. Der Maus-

Sprite wird jedoch automatisch durch die Bewegungen der Maus gesteuert. Mit dem Befehl SPRITE können Sie einen - oder auch mehrere - Sprites definieren und nach Ihren Wünschen auf dem Bildschirm plazieren und bewegen.

Der Aufbau eines solchen Sprites ist dem eines Mauszeigers gleich, denn der Mauszeiger ist ein Sprite. Die Sprite-Form besteht aus einem Raster mit 16*y Pixel, und es wird ein Format-String gebildet, indem man die Sprite-Daten als BitMap-konvertierte-String-Daten aneinanderfügt.

Der Sprite-String hat folgendes Format:

Für jede Grafikzeile des Sprites werden vier Bytes benötigt. Jedes dieser Bytes wird als ASCII-Zeichen in den String abgelegt, wobei die gesetzten Bits einen Punkt markieren und die ungesetzten die freien Lücken. Da ein Sprite eine Breite von 16 Punkten besitzt, werden die ersten und letzten zwei Bytes einer Zeile zu einem Paar zusammengefaßt, das nebeneinander liegende Punkte repräsentiert.

Die zwei Paare sind für die vier verschiedenen Farben zuständig. Es werden nämlich beide 16 Bit breite Punkt-Reihen übereinander gelegt. Für jeden Punkt wird dann das übereinander liegende Bit-Paar untersucht. Es ergeben sich folgende Kombinationen:

Bit-Paar	Farbe
00	transparent
01	erstes Farbregister
10	zweites Farbregister
11	drittes Farbregister

Die Kombination der Bits ergibt einen Farbwert. Der erste Farbwert, die Kombination 00, ist immer transparent, d.h. der Hintergrund bleibt sichtbar. Bei den restlichen drei Einstellungen hängt es von der Sprite-Nummer ab, welche Farbe ihm zugeteilt wird.

Hier ist noch eine Tabelle, bei der Sie erkennen können, welche Farbregister, die Sie bei Bedarf mit SETCOLOR verändern können, bei welchem Sprite die Farbgestalt bestimmen:

Sprite-Nummer	Farbregister
0/1	17, 18, 19
2/3	21, 22, 23
4/5	25, 26, 27
6/7	29, 30, 31

Mit der zweiten Variante läßt sich ganz schnell die Grafik eines Sprites wieder löschen. Somit kann auch der Mauszeiger über seine Nummer (0. Sprite) gelöscht werden.

Die dritte Version erlaubt die Positionierung des Sprites auf dem Screen. Dafür gibt man die X- und Y-Koordinate in einem 320x256-Punkte-Raster an, das unabhängig von der Screen-Auflösung gilt.

Zum Löschen oder Einschalten aller Sprites gibt es noch die Variante mit den Schlüsselwörtern OFF oder ON, die auch den Mauszeiger betreffen. Beispiel:

```
PRINT                              ! Nur, um Ausgabefenster zu aktivieren
sp1$=STRING$(64,-1)
SPRITE #1,sp1$
FOR i%=0 TO 2000 STEP 4            !——.
  x%=108+SIN(i%*PI/180)*(100)          |
  y%=108+COS(i%*PI/180)*(40)           |– Sprite
  VSYNC                                |  darstellen
  SPRITE #1,x%,y%                  !——-.
NEXT i%
SPRITE OFF                         ! Sprite ausschalten
```

VSYNC { VS } VBL-Synchronisation

```
VSYNC
```

Wartet auf den nächsten Strahlrücklauf (Vertikal-Blank). Der Elekronenstrahl beginnt den Bildaufbau in der oberen linken Bildschirmecke und zeichnet dann nacheinander alle Zeilen, bis er in der rechten unteren Ecke angelangt ist. Danach beginnt er den nächsten Aufbau wieder in der linken oberen Ecke. Diesen Weg legt er ca. 50 Mal in der Sekunde zurück

Bei Grafikausgaben mit GET oder PUT kann es sinnvoll sein, den nächsten Bildneuaufbau abzuwarten. Durch die vertikale Synchronisation einer Grafikausgabe mit dem Bildaufbau kann so das Interferenz-Flimmern eingeschränkt werden. Bei Grafikausgaben, die mehr Zeit in Anspruch nehmen, als der Computer zu einem Bildaufbau benötigt, treten allerdings auch dann wieder Interferenzen auf. Ein Beispiel dazu finden Sie unter SPRITE. Die Auswirkung von VSYNC können Sie beobachten, wenn Sie in diesem Beispiel die VSYNC-Zeile löschen.

9.5 Objekt-Animation

Der Amiga ist in der Lage, nicht nur Sprites für bewegte und animierte Darstellung zu verwenden. Zusätzlich werden die sog. BOBs angeboten (BlitterOBjects), die vom Coprozessor "Blitter" verwaltet werden. Der Blitter ist in der Lage, besonders schnell größere Grafik-Flächen zu verschieben und eignet sich so besonders für die Objekt-Animation.

Die vom GFA-BASIC vorgestellten Befehle richten sich vollkommen nach dem AmigaBASIC-Standard, um so ein größtmögliches Maß an Kompatibilität zu erreichen.

OBJECT.SHAPE — Objekt-Aussehen definieren

```
OBJECT.SHAPE ObjNummer, DefString$
OBJECT.SHAPE ObjNummer, AltesObjekt
```

Hier haben wir den Befehl, mit dem Sie das Aussehen eines Objektes bestimmen können. Die erste Syntax-Variante definiert ein Objekt über einen Definitions-String. Diesen String erhalten Sie ganz einfach über ein auf der GFA-Diskette mitgeliefertes Programm. Es heißt IFF_TO_BOB.GFA und erlaubt das Konvertieren von IFF-Brushes, die Sie z.B. mit DeluxePaint erstellen können, in BOBs.

Die zweite Variante kopiert einfach die Gestalt des alten Objektes in das neue. Sie duplizieren damit einfach ein Objekt und können das zweite z.B. mit OBJECT.PLANES in den Farben verändern.

OBJECT.CLOSE — Objekt entfernen

```
OBJECT.CLOSE [ObjNummer [,ObjNummer [,ObjNummer ...]]]
```

Entfernt ein Objekt ganz und gar aus dem Window. Danach ist dieses Objekt auch nicht mehr wieder neu aufrufbar, sondern muß ganz neu definiert werden.

Die Anzahl der zu löschenden Objekte ist beliebig. Trennen Sie mehrere Objekt-Nummern durch Kommata. Sollen alle aktiven Objekte entfernt werden, genügt der Aufruf des Befehls ohne Parameter.

OBJECT.ON Objekt sichtbar machen

```
OBJECT.ON [ObjNummer [,ObjNummer [,ObjNummer ...]]]
```

Schaltet ein zuvor definiertes Objekt ein. Es wird ab jetzt auf dem Bildschirm dargestellt, kann bewegt oder beschleunigt und auch mit allen anderen Befehlen beeinflußt werden.

Auch hier ist die Anzahl der Parameter beliebig. Geben Sie keine Parameter an, werden alle Objekte eingeschaltet.

OBJECT.OFF Objekt unsichtbar machen

```
OBJECT.OFF [ObjNummer [,ObjNummer [,ObjNummer ...]]]
```

Löscht ein Objekt aus dem Window, d.h. die Darstellung des Objektes wird eingestellt. Das Objekt selbst ist aber noch für GFA-BASIC vorhanden, Sie können es also jederzeit wieder neu aktivieren.

Geben Sie durch Kommata getrennt die Nummern der Objekte an, die ausgeschaltet werden sollen. Bei fehlender Angabe werden alle Objekte ausgeschaltet.

OBJECT.CLIP Objekt-Wirkungsbereich festlegen

```
OBJECT.CLIP X_links,Y_oben,X_rechts,Y_unten
```

Die von GFA-BASIC unterstützten Objekte müssen nicht im gesamten Bereich des Windows zugelassen sein. Sie können auch einen Bereich explizit definieren, der den Freiraum der BOBs einschränkt. Dafür wird OBJECT.CLIP benutzt. Die Koordinatenangaben verstehen sich absolut innerhalb des Windows.

OBJECT.START — Objekt-Bewegung starten

```
OBJECT.START [ObjNummer [,ObjNummer [,ObjNummer ...]]]
```

Startet die Bewegung der angegebenen Objekte. Fehlt die spezielle Angabe von Objekt-Nummern, bezieht sich der Befehl auf alle aktiven Objekte.

OBJECT.STOP — Objekt-Bewegung anhalten

```
OBJECT.STOP [ObjNummer [,ObjNummer [,ObjNummer ...]]]
```

Stoppt die Bewegung der angegebenen Objekte. Fehlt die explizite Angabe einer oder mehrerer Objekt-Nummern, so werden alle Objekte angehalten.

OBJECT.X — X-Position bestimmen

```
OBJECT.X ObjNummer, X_Position
```

Bestimmt die X-Position des Objektes. Dieser Wert bezieht sich auf das Window, in dem das Objekt dargestellt wird.

OBJECT.Y — Y-Position bestimmen

```
OBJECT.Y ObjNummer, Y_Position
```

Bestimmt die Y-Position des Objektes. Dieser Wert bezieht sich auf das Window, in dem das Objekt dargestellt wird.

OBJECT.AX — Objekt beschleunigen

```
OBJECT.AX ObjNummer, PixSec
```

Wenn Sie ein Objekt in Ihrem Programm bewegen wollen, dann läßt sich dies nicht nur linear machen (das Objekt bewegt sich ständig mit der gleichen Geschwindigkeit), sondern Sie können diese Grafik auch beschleunigen. Dazu wird dieser Befehl verwendet, bei dem Sie unter Angabe der Objekt-Nummer eine Beschleunigungsrate angeben können, die den Geschwindigkeitszuwachs in Pixeln pro Sekunde definiert.

OBJECT.AX legt die Beschleunigung des Objektes alleine in X-Richtung fest!

OBJECT.AY — Objekt beschleunigen

```
OBJECT.AY ObjNummer, PixSec
```

Zur Beschleunigung eines Objekts in Y-Richtung. Sehen Sie für eine Beschreibung unter OBJECT.AX nach.

OBJECT.VX — Objekt-Geschwindigkeit festlegen

```
OBJECT.VX ObjNummer, X_Geschwindigkeit
```

Bestimmt die Geschwindigkeit des Objektes in X-Richtung. Diese wird in Pixeln pro Sekunde angegeben.

OBJECT.VY — Objekt-Geschwindigkeit festlegen

```
OBJECT.VY ObjNummer, Y_Geschwindigkeit
```

Bestimmt die Geschwindigkeit des Objektes in Y-Richtung. Diese wird in Pixeln pro Sekunde angegeben.

OBJECT.PLANES — Farbeinstellung des Objekts festlegen

```
OBJECT.PLANES ObjNummer [,Bitebenen [,Ebenenwert]]
```

Dieser sehr interne Befehl erlaubt es, die Darstellung des BOBs zu beeinflussen. Dabei werden zwei 8-Bit-Werte angegeben, die die jeweilige Verteilung der Grafik auf die einzelnen BitPlanes des Screen bestimmen.

Mit der Einstellung Bitebenen geben Sie an, in welche BitPlanes die BOB-Grafik geschrieben werden soll. Setzen eines Bits bedeutet, daß die vom BOB für diese BitPlane vorgesehene Grafik dargestellt werden soll. Löschen Sie ein Bit, obwohl dafür eine BitPlane definiert wurde, so verliert der BOB an dieser Stelle seine Farbe.

Ebenenwert gleicht die oben ausgeschalteten BitPlanes wieder aus. Indem Sie dort gelöschte Bits hier setzen, tragen Sie dem Amiga auf, diese Bitplanes ganz zu setzen, daß heißt dort eine Farbe darzustellen.

So können Sie mit dem ersten Wert Farben aus der Grafik entfernen und mit dem zweiten Farben dazufügen. Sie können aber auch ein BOB mit nur einer BitPlane nehmen, und es durch Zuschalten nicht benutzter BitPlanes im Ebenenwert einfärben. Damit lassen sich ganz einfach verschiedenfarbige BOBs mit gleicher interner Definition erstellen.

Hier noch ein Rechenbeispiel, das Ihnen die Bestimmung der beiden Werte erleichtert:

Das BOB hat vier BitPlanes, von denen nur die erste und die dritte BitPlane dargestellt werden sollen.

BitPlane 1 : 2^0 = 1
BitPlane 2 : 2^1 = 2
BitPlane 3 : 2^2 = 4
BitPlane 4 : 2^3 = 8
BitPlane 5 : 2^4 = 16
BitPlane 6 : 2^5 = 32

Wir bekommen also für Bitebenen einen Wert von 5. Außerdem soll noch die vierte BitPlane bei der Darstellung mit Bits gefüllt werden. Dazu wird der Wert 8 bei Ebenenwert eingetragen.

OBJECT.PRIORITY — Objekt-Reihenfolge einstellen

```
OBJECT.PRIORITY ObjNummer, Priorität
```

Sobald mehrere BOBs in einem Window dargestellt werden, tritt für den Computer das Problem auf, daß er nicht weiß, in welcher Reihenfolge er die BOBs zeichnen soll. Das ist besonders für eine perspektivische Darstellung sehr wichtig, denn dort lebt der Effekt von Überlappungen, die Teile der Grafik verdecken.

Sie können mit diesem Befehl für jeden BOB eine Priorität festlegen, mit der er gezeichnet wird. Je höher die Priorität, desto später wird das Objekt gezeichnet, es liegt also später über anderen Objekten mit geringerer Priorität.

Der Wert dafür kann zwischen 0 und 32767 liegen und reicht damit bestimmt für Ihre Anwendungen aus.

OBJECT.HIT — Objekt-Kollision Auswahl treffen

```
OBJECT.HIT ObjNummer [, Wert1 [,Wert2]]
```

Manchmal ist es nötig, daß die Kollision zweier Objekte nicht registriert wird. Dazu könnte man einerseits in der Abfrage bei der Untersuchung manche Fälle außer acht lassen. Dies erweist sich aber nicht als günstig, weil für jeden anderen Fall immer wieder neue Abfragen geschrieben werden müssen.

Deshalb gibt es den Befehl OBJECT.HIT, mit dem die Beziehung aller Objekte neu definiert werden kann. Unter Beziehung verstehen wir, ob eine Kollision angezeigt oder nicht angezeigt werden soll. Dabei wird wieder auf verschiedene Bitmuster zurückgegriffen, die angeben, welche Objektnummern bei einer Kollision gemeldet werden sollen.

GFA-BASIC meldet alle Kollisionen. Wir verändern das durch die beiden Werte. Der erste, ein 16-Bit-Wert, bestimmt mit jedem seiner Bits, mit welchen anderen Objekten es zusammenstoßen kann, so daß eine Meldung erfolgt. Bei einer Kollision wird einfach der Wert2 des fremden Objektes mit dem Wert1 unseres Objektes verknüpft. Nur wenn ein Wert ungleich null entsteht, wird eine Kollision gemeldet.

OBJECT.X() — X-Position ermitteln

```
X_Position=OBJECT.X(ObjNummer)
```

Bestimmt die aktuelle X-Position des über ObjNummer identifizierten Objektes.

OBJECT.Y() Y-Position ermitteln

```
Y_Position=OBJECT.Y(ObjNummer)
```

Bestimmt die aktuelle Y-Position des über ObjNummer identifizierten Objektes.

OBJECT.VX() Geschwindigkeit in X-Richtung ermitteln

```
X_Geschwindigkeit=OBJECT.VX(ObjNummer)
```

Bestimmt die momentane Geschwindigkeit in X-Richtung in Pixeln pro Sekunde.

OBJECT.VY() Geschwindigkeit in Y-Richtung ermitteln

```
Y_Geschwindigkeit=OBJECT.VY(ObjNummer)
```

Bestimmt die momentane Geschwindigkeit in Y-Richtung in Pixeln pro Sekunde.

OBJECT.AX() Beschleunigung in X-Richtung ermitteln

```
X_Beschleunigung=OBJECT.AX(ObjNummer)
```

Bestimmt die aktuelle Beschleunigung in X-Richtung in Pixeln pro Sekunde.

OBJECT.AY() Beschleunigung in Y-Richtung ermitteln

```
Y_Beschleunigung=OBJECT.AY(ObjNummer)
```

Bestimmt die aktuelle Beschleunigung in Y-Richtung in Pixeln pro Sekunde.

ON COLLISION GOSUB Bei Objektkollision verzweigen

```
ON COLLISION GOSUB Prozedur
```

Legt die Prozedur fest, die bei Kollision zweier Objekte oder eines Objektes mit dem Window-Rand angesprungen werden soll.

COLLISION() Kollisionssart feststellen

```
Ort=COLLISION(ObjNummer)
```

Stelle fest, mit welchem Objekt das angegebene zusammengestoßen ist. Dabei können vier Sonderfälle eintreten:

Ergebnis	Bedeutung
-1	oberer Window-Rand
-2	linker Rand
-3	unterer Rand
-4	rechter Rand

Außerdem kann man als Argument die Werte 0 oder -1 angeben. Dabei liefert COLLISION(0) die Nummer des Objekts, das als letztes an der Kollision beteiligt war, und COLLISION(-1) liefert die Nummer des Windows, in dem die Kollision stattfand.

10. Datenumwandlung

ASC() — Textzeichen => ASCII-Wert

```
Var=ASC("Zeichen")
```

Ermittelt den ASCII-Wert des Textzeichens 'Zeichen'. Bei Strings wird nur der ASCII-Wert des ersten Zeichens zurückgegeben. Ist der angegebene String leer (""), wird der Wert 0 geliefert. ASC() bildet die Umkehrfunktion zu CHR$().

Eine Tabelle der möglichen Zeichen und ihrer ASCII-Werte finden Sie im Anhang. ASCII: American Standard Code for Information Interchange (Deutsch: Amerikanischer Standard-Code für Informationsaustausch).

BIN$() — Numerisch => Binär

```
Var$=BIN$(Expr)
Var$=BIN$(Expr [,Stellen])
```

Wandelt Expr in einen Binär-String um. Expr steht für eine(n) beliebige(n) numerische(n) Variable, Konstante, Ausdruck oder Funktion.

Durch den optionalen Parameter Stellen kann eine Stellenanzahl (1 - 32) vorgegeben werden, auf die der gewandelte Wert begrenzt wird. Beispiele:

```
Print Bin$(1273530)
'
Print
For I%=1 To 4             ! 4 Binärwerte
  Read A%                 ! Lesen
  Print Bin$(A%),"="'A%   ! Wandeln und ausgeben
```

```
Next I%
Data &X1001110,&X100110101,&X10001110,&X100111001
'
Print
For I%=1 To 12                ! 12 Binärwerte
  Read A$                     ! Als String lesen
  B$=Right$(String$(14,"0")+A$,14) ! Binär-String auf
  '                           ! 14 Zeichen formatieren
  Print B$,                   ! Binär-String ausgeben
  Print "="'Val("&X"+A$) ! Wert ausgeben
  For J%=1 To Len(B$)         ! Alle Zeichen des Strings
    If Val("&X"+B$) And 2^J% ! Bit gesetzt?
      Plot 200+J%,30+I%       ! Dann Punkt setzen
    Endif
  Next J%
Next I%
Data 000111000111000111000
Data 000111000111000111000
Data 000111000111000111000
Data 111000111000111000111
Data 111000111000111000111
Data 111000111000111000111
Data 000111000111000111000
Data 000111000111000111000
Data 000111000111000111000
Data 111000111000111000111
Data 111000111000111000111
Data 111000111000111000111
```

10.1 Die Zahlensysteme

Es gibt im Computerbereich vier Arten von Zahlensystemen:

- Das dezimale System
- Das hexadezimale System
- Das binäre System
- Das oktale System

Das allgemein übliche Dezimalsystem hat die Zahl 10 zur Basis (dezi: von lat. decem => zehn/z.B. Dezimeter = 10 Zentimeter). Diese Basis wird jeweils zur Ermittlung der Wertigkeit einer Zahl herangezogen. So werden die Einerstellen aus der Null-Potenz der Zahl 10 ermittelt (10 hoch 0 = 1), die Zehnerstellen aus

der Einer-Potenz (10 hoch 1 = 10), die Hunderterstellen aus der Zweier-Potenz (10 hoch 2 = 100) usw.

In den anderen Zahlensystemen ist dieser Potenzierungsvorgang exakt derselbe, nur werden hier andere Zahlen als Basis verwendet. So wird im Hexadezimalsystem (Hexa + Dezi = Hexadezi/6 + 10 = 16) die Zahl 16 als Basis verwendet, im Binärsystem (Bi = 2) die Zahl 2 und im Oktalsystem (Okta = 8) die Zahl 8. So erklären sich die geringen Wertigkeiten der Binärzahlen in ihren Stellen und die hohen Wertigkeiten der Hexadezimalzahlen.

Stellenwertigkeiten:

Dezimal:

<-... 5.	4.	3.	2.	1.	Stelle
<-... 10000	1000	100	10	1	Wert/Format
<-... 10^4	10^3	10^2	10^1	10^0	Potenz

Hexadezimal:

<-... 5.	4.	3.	2.	1.	Stelle
<-... 65536	4096	256	16	1	Wert
<-... $10000	$1000	$100	$10	$1	Format
<-... 16^4	16^3	16^2	16^1	16^0	Potenz

Binär:

<-... 5.	4.	3.	2.	1.	Stelle
<-... 16	8	4	2	1	Wert
<-... %10000	%1000	%100	%10	%1	Format
<-... 2^4	2^3	2^2	2^1	2^0	Potenz

Oktal:

	4.	3.	2.	1.	
<-... 5.	4.	3.	2.	1.	Stelle
<-... 4096	512	64	8	1	Wert
<-... &10000	&1000	&100	&10	&1	Format
<-... 8^4	8^3	8^2	8^1	8^0	Potenz

Bei den Hexadezimalzahlen ist eine Besonderheit zu beachten. Da sich die Zahl 16 mit arabischen Zahlen nicht einstellig darstellen läßt, hat man zu einem Trick gegriffen. Die Hex-Zahlen 0 - 9 werden genauso dargestellt wie im Dezimalsystem. Die Zahlen von 10 bis 15 werden dagegen durch einen Buchstaben repräsentiert (A=10;B=11;C=12; D=13;E=14;F=15). Daher also die Buchstaben in Hexadezimalzahlen.

Die Dezimalzahl 1037 setzt sich also wie folgt zusammen:

```
  (10 hoch 0) * 7 =     7
+ (10 hoch 1) * 3 =    30
  (die dritte Stelle ist nicht besetzt, also: 0)
+ (10 hoch 3) * 1 =  1000
                  ----------
     Summe :         1037
                  ==========
```

Die Binärzahl %100011 ergibt dagegen nach der Wertigkeit ihrer Stellen (immer von rechts gesehen, wie im Dezimalsystem auch):

```
  (2 hoch 0) * 1  = 1
+ (2 hoch 1) * 1  = 2
  (die dritte Stelle ist nicht besetzt, also: 0)
  (die vierte Stelle ist nicht besetzt, also: 0)
  (die fünfte Stelle ist nicht besetzt, also: 0)
+ (2 hoch 5) * 1  = 32
                 ----------
     Summe :        35
                 ==========
```

Das gleiche für die Hexadezimalzahl $F3B0:

```
      (die erste Stelle ist nicht besetzt, also: 0)
      (16 hoch 1) * 11 =   176
    + (16 hoch 2) * 3  =  2816
    + (16 hoch 3) * 15 = 61440
                       ----------
         Summe :          64432
                       ==========
```

Und noch für die Oktalzahl &10537:

```
      (8 hoch 0) * 7   =    7
    + (8 hoch 1) * 3   =   24
    + (8 hoch 2) * 5   =  320
      (die vierte Stelle ist nicht besetzt, also: 0)
    + (8 hoch 4) * 1   = 4096
                       ----------
         Summe :          4447
                       ==========
```

Die Umkehrung dieser Transformation sieht dann so aus: Die Dezimalzahl 1352 soll in das binäre Format verwandelt werden:

```
1352 : 2   =   676   Rest 0   >-----------------------.
 676 : 2   =   338   Rest 0   >---------------------. |
 338 : 2   =   169   Rest 0   >-------------------. | |
 169 : 2   =    84   Rest 1   >-----------------. | | |
  84 : 2   =    42   Rest 0   >---------------. | | | |
  42 : 2   =    21   Rest 0   >-------------. | | | | |
  21 : 2   =    10   Rest 1   >-----------. | | | | | |
  10 : 2   =     5   Rest 0   >---------. | | | | | | |
   5 : 2   =     2   Rest 1   >-------. | | | | | | | |
   2 : 2   =     1   Rest 0   >-----. | | | | | | | | |
   1 : 2   =     0   Rest 1   >---. | | | | | | | | | |
   0 : 2   =     0   Rest 0   >-. | | | | | | | | | | |
                                | | | | | | | | | | | |
        Die Binärzahl lautet:   0 1 0 1 0 1 0 0 1 0 0 0
```

Nach dem gleichen Schema lassen sich auch Dezimalzahlen in die übrigen beiden Zahlensysteme konvertieren. Dazu noch das Beispiel für Hexadezimalzahlen: Die Dezimalzahl 35117 soll in das Hexadezimalformat gewandelt werden:

```
35117 : 16 = 2194 => 2194 * 16 = 35104 => Diff. = 13 (D) >-----.
 2194 : 16 =  137 =>  137 * 16 =  2192 => Diff. =  2 (2) >---. |
  137 : 16 =    8 =>    8 * 16 =   128 => Diff. =  9 (9) >-. | |
```

```
8 : 16 =    0 =>    0 * 16 =    0 => Diff. =  8 (8) ──┐│││
0 : 16 =    0 =>    0 * 16 =    0 => Diff. =  0       ││││
                                                      ││││
        Die Hexadezimalzahl lautet :                  892D
```

Bei derart komfortablen Sprachen wie GFA-BASIC hat man es natürlich nicht mehr nötig, diese Zahlen und Formate "zu Fuß" zu ermitteln. Trotzdem kann es unter Umständen von Vorteil sein, sich damit einigermaßen auszukennen.

Das hexadezimale Zahlensystem wurde aus einem bestimmten Grund entwickelt. Es lassen sich nämlich "zufälligerweise" genau die Inhalte von vier Bits (Tetrade oder auch Nibble) mit einer Hex-Zahl darstellen. Eine vierstellige Binärzahl kann maximal den Wert 15 (2^0+2^1+2^2+2^3) annehmen. Und genau dieser Wert läßt sich auch maximal mit einer Hex-Ziffer darstellen (F). Ein Byte (8 Bit) wird demnach immer zu einer 2er-Tetrade (2*4 = 8 Bit), ein Word (16 Bit) zu einer 4er-Tetrade (4*4 = 16 Bit) und ein Longword (32 Bit) immer zu einer 8er-Tetrade (8*4 = 32 Bit) zusammengefaßt.

CFLOAT() — Integerwert => Fließkommawert

```
Var=CFLOAT(Wert)
```

Wandelt Wert (Integerwert) in eine Realzahl um. CFLOAT bildet die Umkehrfunktion zu CINT.

CHR$() — ASCII => Textzeichen

```
Var$=CHR$(Wert)
```

Liefert ein - dem angegebenen Wert entsprechendes - ASCII-Zeichen. Ist Wert größer als 255, so wird das Zeichen ermittelt,

das dem Wert MOD 256 (bzw. Wert AND 255) entspricht. CHR$ bildet die Umkehrfunktion zu ASC.

Beispiele hierzu finden Sie z.B. unter INKEY$(), LINE INPUT, PRINT, WRITE und ASC().

CINT() Fließkommawert => Integerwert

```
Var=CINT(Wert)
```

Wandelt Wert (Realwert) in eine Integerzahl um. Im Gegensatz zu INT wird die Zahl vorher exakt gerundet. CINT bildet die Umkehrfunktion zu CFLOAT.

CVI(), CVL(), CVS(), CVD() String => Format-Zahl

```
Funktion: Ergebnis:
Var=CVI("2 Zeichen") 16-Bit-Integerwert
Var=CVL("4 Zeichen") 32-Bit-Integerwert
Var=CVS("4 Zeichen") Realwert (Amiga-BASIC-Format)
Var=CVD("8 Zeichen") Realwert (MBASIC- oder GFA-BASIC-Format)
```

Es wird die der jeweiligen Funktion entsprechende Zeichenanzahl von x Zeichen in eine Zahl des jeweiligen Formats umgewandelt. Diese Funktionen bilden die Umkehrfunktionen zu MKI$/MKL$/MKS$/MKD$.

Beispiele finden Sie unter INSTR() und in der Prozedur Cut unter RIGHT$().

HEX$() Numerisch => Hexadezimal

```
Var$=HEX$(Expr)
Var$=HEX$(Expr [,Stellen])
```

Wandelt Expr in einen Hexadezimal-String um. Expr steht für eine(n) beliebige(n) numerische(n) Variable, Konstante, Ausdruck oder Funktion. Durch den optionalen Parameter Stellen kann eine Stellenanzahl (1 - 8) vorgegeben werden, auf die der gewandelte Wert begrenzt wird.

Außerdem wird das Standard-Präfix für Hexadezimalzahlen (z.B. $FA5C16) erkannt, und - zulässige - Wertangaben dieser Art werden vom Interpreter selbständig in das GFA-Format (z.B. &HFA5C16) umgewandelt. Wird als Präfix nur das Und-Zeichen & (z.B. &1EA6F9) verwendet, wird der Ausdruck ebenfalls in den entsprechenden Hex-Wert umgewandelt.

MKI$(), MKL$(), MKS$(), MKD$() Format-Zahl => String

```
Funktion: Ergebnis:
Var$=MKI$(16-Bit-Integer-Wert) 2-Zeichen-String
Var$=MKL$(32-Bit-Integer-Wert) 4-Zeichen-String
Var$=MKS$(Amiga-BASIC-Realwert) 4-Zeichen-String
Var$=MKD$(MBASIC- oder GFA-BASIC-Realwert) 8-Zeichen-String
```

Es wird der in Klammern angegebene Wert in einen der Wertgröße und dem gewünschten Format entsprechenden String-Ausdruck umgewandelt.

Der Variablenaufbau der verschiedenen Systeme, Interpreter und Compiler kann sich stark voneinander unterscheiden. Um nicht zeitaufwendige Rechenoperationen ausführen zu müssen, um die einzelnen Werte anderer Sprachen in das benötigte Format zu übertragen, können diese Funktionen auch dazu verwendet werden, den Datenaustausch zu vereinfachen.

Diese Funktionen bilden die Umkehrfunktionen zu CVI()/ CVL()/CVS()/CVD().

Beispiele hierzu finden Sie unter anderem in der Prozedur Cut unter RIGHT$().

OCT$() Numerisch => Oktal

```
Var$=OCT$(Expr)
Var$=OCT$(Expr [,Stellen])
```

Wandelt Expr in einen Oktal-String um. Expr steht für eine(n) beliebige(n) numerische(n) Variable, Konstante, Ausdruck oder Funktion. Will man Integerwerte im Oktal-Format angeben, so kann der Vorsatz &O (z.B.: A%=&O16501) verwendet werden.

Durch den optionalen Parameter Stellen kann eine Stellenanzahl (1 - 11) vorgegeben werden, auf die der gewandelte Wert begrenzt wird.

Weitere Informationen finden Sie in Kapitel 10.1 "Die Zahlensysteme".

STR$() Numerisch => String

```
Var$=STR$(Wert)
Var$=STR$(Wert [,Stellen [,Real]])
```

Es wird ein Text-String mit der Länge gebildet, die der Anzahl der Ziffern des übergebenen Wertes im Dezimalformat entspricht. Wert kann in jedem beliebigen Zahlensystem angegeben werden. Als Hexadezimal-, Binär- oder Oktalzahl angegebene Werte werden vorher in das Dezimalformat umgewandelt.

Die hiermit erzeugte Ziffernfolge ist keine Zahl mehr, die einen Wert darstellt, sondern lediglich ein String, der die einzelnen Ziffern des Wertes als Textzeichen enthält. STR$ bildet die Umkehrfunktion zu VAL.

Durch den optionalen Parameter Stellen kann eine Stellenanzahl vorgegeben werden (Vor-, Nachkommastellen und ggf. Dezimalpunkt), auf die der gewandelte Wert begrenzt wird. Der optionale Parameter Real gibt an, auf wie viele Nachkommastellen der gewandelte Wert ggf. gerundet werden soll. Diese gerundeten Nachkommastellen gehen auf jeden Fall in den gelieferten Wert-String ein, auch wenn Wert eigentlich keine Nachkommastellen beinhaltet. Beispiele:

```
PRINT STR$(572.6169,5,3)   ergibt:   2.617
PRINT STR$(6169,9,5)       ergibt:  169.00000
```

VAL() — String => Numerisch

```
Var=VAL(Var$)
```

Wandelt alle am Anfang eines Strings stehenden Zeichen, die sich zur Darstellung numerischer Werte eignen, in eine dezimale Realzahl um.

Var$ ist eine beliebige Zeichenkette, ein String-Ausdruck oder eine String-Variable, deren Inhalt vom Anfang ausgehend daraufhin untersucht wird, ob Textzeichen enthalten sind, die einen Wert in einem der vier Zahlensysteme darstellen. Die Suche wird abgebrochen, wenn das String-Ende erreicht ist oder die Funktion auf ein Textzeichen trifft, das nicht wandelbar ist.

Ist das erste Zeichen des Strings ein nicht wandelbares Textzeichen oder ist der String leer, wird eine Null zurückgegeben. Beispiele:

```
A$="> ";Str$(123456);" <"
Print A$                    ergibt: > 123456 <
'
A$="1011010 <-Binär"
Print Val("&X"+A$)          ergibt: 90        (= &X1011010)
'
A$="1141331 <-Octal"
Print Val("&O"+A$)          ergibt: 312025    (= &O1141331)
'
A$="AF451DE <-Hexadezimal"
Print Val("&H"+A$)          ergibt: 183783902 (= &HAF451DE)
Print Val("&HEEZeichen")    ergibt: 238             (= &HEE)
'
A$="&X110123"
A%=Val(A$)
Print A%                    ergibt: 13           (= &X1101)
'
Print Val("2.37E+07")       ergibt: 23700000  (= 2.37E+07)
```

VAL?() — Anzahl wandelbarer Textzeichen ermitteln

```
Var=VAL?(Var$)
```

Ermittelt ab Anfang von Var$ die Anzahl seiner Zeichen, die in numerische Werte konvertiert werden können (siehe VAL()). Var$ steht für eine beliebige Zeichenkette oder String-Variable, die auf die Anzahl ihrer wandelbaren Zeichen untersucht werden soll. Trifft die Funktion auf nicht wandelbare Zeichen, wird die Untersuchung abgebrochen. Beispiele:

```
Print Val?("237E07")     ergibt: 6  (Exponentialformat)
'
A$="&X110123E"
Print Val?(A$)           ergibt: 6  (inkl. Identifikator)
```

11. Feld-, Speicher- und Zeigeroperationen

11.1 Feldoperationen

ARRAYFILL { ARR } — Feld mit Wert belegen

```
ARRAYFILL Feld(),Var
```

'Feld' bezeichnet ein bereits beliebig dimensioniertes numerisches Feld (Integer, Real, Boole). Alle Elemente dieses Feldes werden mit dem Wert Var belegt. Var muß mit dem Feldtyp übereinstimmen (z.B. Integer zu Integer).

Beispiel:

Umständlich: Dim Feld%(20,32,16)

```
For A%=0 To 20             !——.  Füllt alle
  For B%=0 To 32              |  Elemente des
    For C%=0 To 16            |- Arrays Feld%
      Feld%(A%,B%,C%)=237     |  mit dem Wert
    Next C%                   |  237.
  Next B%                     |
Next A%                    !——'
GFA-Methode:  Dim Feld%(20,32,16)
Arrayfill Feld%(),237  !——  Tut dasselbe
```

DELETE { DEL } — Einzelelement aus Feld löschen

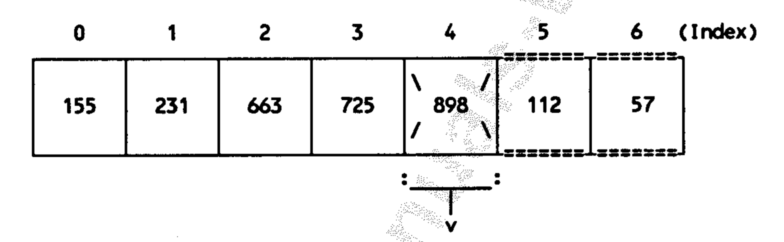
```
DELETE Feld(Index)
DELETE Feld$(Index)
```

Löscht das einzelne Element 'Index' aus dem Feld 'Feld' bzw. 'Feld$'. Alle darüberliegenden Elemente werden im Feld um eine Stelle nach unten versetzt. Das letzte Element enthält an-

schließend den Wert Null bzw. bei String-Feldern einen Leer-String. DELETE ist die Umkehrung zu INSERT.

Beispiel:

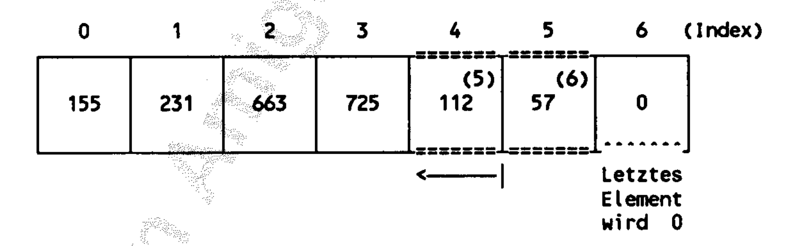
Vorher:



Dann:

```
DELETE Feld(4)
```

Nachher:



DIM — Feld(er) dimensionieren

```
DIM Arr1(Ind1 [,Ind2,...]) [,Arr2(Ind1 [,Ind2,...])...]
```

Legt die Dimension(en) von Arr1 (bzw. Arr2, Arr3 etc.) fest und reserviert hierfür Speicherplatz. 'Arr' steht für beliebige numerische oder alphanumerische Felder.

'Ind' besagt, wie viele Elemente pro Dimension zugelassen sind. Bei mehrdimensionalen Feldern (z.B. DIM Feld(5,20,7)) ist die Anzahl der Elemente auf 65535, bei eindimensionalen Feldern (z.B. DIM Feld$(100)) nur durch die Größe des Arbeitsspeichers begrenzt.

Achtung: Bei großen Dimensionierungen kann sich die Adressenlage der übrigen Variablen - insbesondere bei String-Variablen - verschieben. Falls Maschinenprogramme in String-Variablen abgelegt wurden und das Programm ohne vorherige VARPTR-Abfrage aufgerufen wird, führt dies ggf. zum Absturz. Zur Speicherung der Routine verwendet man daher besser INLINE (siehe dort).

11.1.1 Aufbau eines mehrdimensionalen Feldes

Stellen Sie sich bitte einen Schrank vor. Dieser Schrank wird nun in zwei Hälften geteilt, und zwar in eine rechte und eine linke Hälfte. Innerhalb der Hälften sind Schubladen untergebracht. Nehmen wir an, jede Hälfte besitzt zwei Schubladen.

Diese Schubladen werden nun wiederum in einzelne Fächer unterteilt. Jede Schublade erhält drei Fächer. Wir haben nun also einen Schrank mit 12 Fächern (2*2*3) eingerichtet.

Die entsprechende Dimensionierung dazu:

```
DIM Schrank(1,1,2)  -> bei OPTION BASE 0
DIM Schrank(2,2,3)  -> bei OPTION BASE 1
```

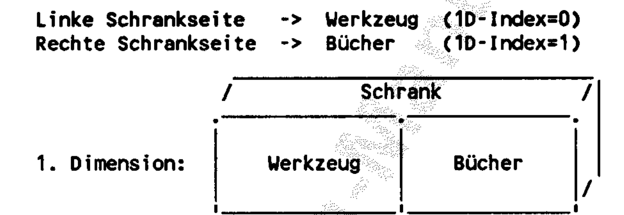
Sie wundern sich evtl., warum im ersten Fall nicht (2,2,3) steht. Das hat den Grund, daß Arrays immer mit dem Index 0 zu zählen beginnen, falls nicht durch OPTION BASE 1 das Null-Element eliminiert wurde. So wird im ersten Fall vorausgesetzt, daß als jeweils kleinster Index einer Dimension ein Null-Element vorhanden ist.

Der angegebene Index bedeutet dann: "Dimensioniere bis Element X", also:

```
Element 0 und 1 der ersten Dimension      --.
Element 0 und 1 der zweiten Dimension       |-> DIM(1,1,2)
Element 0, 1 und 2 der dritten Dimension --'
```

Im folgenden gehe ich grundsätzlich davon aus, daß OPTION BASE 0 aktiv ist. In diesem Schrank sollen nun verschiedene "Dinge" untergebracht werden. In unserem Fall sind dies Zahlen (hier: Mengenwerte). Zur besseren Orientierung ordne ich den beiden Schrankseiten erst einmal Begriffe zu:

```
Linke Schrankseite   -> Werkzeug  (1D-Index=0)
Rechte Schrankseite  -> Bücher    (1D-Index=1)
```

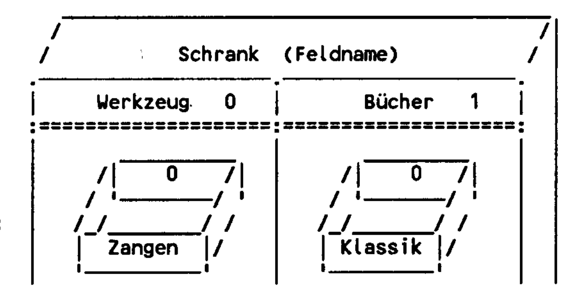


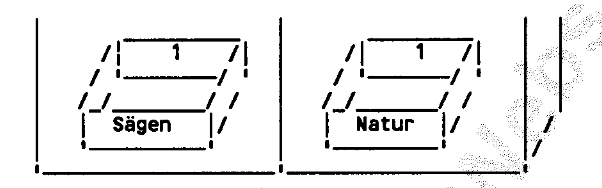
Den Schubladen der Werkzeug-Seite gebe ich die Aufschriften:

```
1. Schublade  -> Zangen  (2D-Index=0)
2. Schublade  -> Sägen   (2D-Index=1)
```

Die Schubladen der Bücher-Seite bekommen die Namen:

```
1. Schublade  -> Klassik (2D-Index=0)
2. Schublade  -> Natur   (2D-Index=1)
```





Nun gebe ich den Elementen einen Namen:

```
.___________________________________________________________________.
| Auf der 'Werkzeug'-Seite              (Index 0 der 1.Dimension)  |
'-.________________________________________________________________'
  |
  |    .________________________________________________________.
  |  .->| in der Schublade 'Zangen' (Index 0 der 2.Dimension)   |
  |  |  '--.____________________________________________________'
  |  |     |
  |  |     |    .___________________________________________.
  |  |     |  .->| 1. Fach    ->  'Kneifzangen'  (3D-Index=0) |
  |  |     |  |  :===========================================:
  '--|     '-|->| 2. Fach    ->  'Rohrzangen'   (3D-Index=1) |
     |        |  :===========================================:
     |        '->| 3. Fach    ->  'Spitzzangen'  (3D-Index=2) |
     |           '___________________________________________'
     |
     |  .________________________________________________________.
     '->| in der Schublade 'Sägen'  (Index 1 der 2.Dimension)   |
        '--.____________________________________________________'
           |
           |    .___________________________________________.
           |  .->| 1. Fach    ->  'Kettensägen'  (3D-Index=0) |
           |  |  :===========================================:
           '-|->| 2. Fach    ->  'Bügelsägen'   (3D-Index=1) |
              |  :===========================================:
              '->| 3. Fach    ->  'Stichsägen'   (3D-Index=2) |
                 '___________________________________________'

.___________________________________________________________________.
| Auf der 'Bücher'-Seite                (Index 1 der 1.Dimension)  |
'-.________________________________________________________________'
  |
  |    .________________________________________________________.
  |  .->| in der Schublade 'Klassik'(Index 0 der 2.Dimension)   |
  |  |  '--.____________________________________________________'
  |  |     |    .___________________________________________.
  |  |   .->| 1. Fach    ->  'Essays'       (3D-Index=0) |  |   |
  |  |   |  :===========================================:
  '--|     '-|->| 2. Fach    ->  'Novellen'     (3D-Index=1) |
     |        |  :===========================================:
     |        '->| 3. Fach    ->  'Romane'       (3D-Index=2) |
     |           '___________________________________________'
     |
     |  .________________________________________________________.
     '->| in der Schublade 'Natur'  (Index 1 der 2.Dimension)   |
        '--.____________________________________________________'
```

```
|    .______________________________________________.
| .->| 1. Fach     ->  'Tiere'        (3D-Index=0) |
|    :==============================================:
'-|->| 2. Fach     ->  'Pflanzen'     (3D-Index=1) |
  |  :==============================================:
  '->| 3. Fach     ->  'Steine'       (3D-Index=2) |
     '______________________________________________'
```

Bis hierhin sieht das vielleicht etwas übertrieben aus. Der Sinn der Sache ist aber der, daß nun anhand von sogenannten "Indizes" auf jedes einzelne Fach der untersten Ebene zugegriffen werden kann.

Unter Index versteht man allgemein ein Unterscheidungsmerkmal bzw. eine Kennzeichnung gleichartiger Größen, die dann statt durch ihren Namen durch einen ihnen zugewiesenen Tabellenwert (Tabellenposition = Index) identifiziert werden können, z.B.:

Oberbegriff 'Tier' 1 = 'Haus-'
2 = 'Nutz-'
3 = 'Raub-'
4 = 'Schalen-'
5 = 'Stachel-'

Statt des Begriffs 'Raubtier' könnte man nun auch 'Tier(3)' sagen und dann - um die Bedeutung von (3) zu erfahren - in der Tabelle unter dem Index 3 nachschauen.

Stellt man sich für die Elemente 'Pflanzen', 'Romane' usw. in der untersten Dimension Speicherplätze im Computer vor, so kann man sich nun durch die Angabe der Indizes jederzeit Informationen über den Inhalt der Plätze verschaffen. Dieses ist vor allem dann wichtig, wenn Informationen abgelegt werden, die bestimmten 'Familien' zugeordnet werden sollen.

Um nun z.B. den Inhalt des Faches "Pflanzen" zu ermitteln, kann man durch Ausgaben (z.B. PRINT Schrank(1,1,0)), eine beliebige Zuweisung (z.B. A%=Schrank(1,1,0)) oder Einbindung in einen Ausdruck (z.B. IF 2*Schrank(1,1,0)+10) dieses Fach lesen oder durch Schrank(1,1,0)=XYZ etwas darin ablegen.

Um die Identifikation der einzelnen Schrankseiten, Schubladen und der darin enthaltenen Fächer zu vereinfachen, kann man den numerischen Indizes auch Variablennamen zuweisen:

- Werkzeug=0
  - Zangen=0
    - Kneifzangen=0
    - Rohrzangen=1
    - Spitzzangen=2
  - Saegen=1
    - Kettensaegen=0
    - Buegelsaegen=1
    - Stichsaegen=2
- Bücher=1
  - Klassik=0
    - Essays=0
    - Novellen=1
    - Romane=2
  - Natur=1
    - Tiere=0
    - Pflanzen=1
    - Steine=2

Jetzt fällt es natürlich leicht, z.B. gezielt in Erfahrung zu bringen, wie viele Bücher zum Thema "Pflanzen" in meiner Bibliothek stehen:

```
IF Schrank(Buecher,Natur,Pflanzen)=0
  PRINT "Natur-Banause!"
ENDIF
```

oder:

```
IF Schrank(Werkzeug,Zangen,Rohrzangen)>10
  PRINT "Na dann: fröhlichen Rohrbruch!"
ENDIF
```

Ich hoffe, daß Ihnen dieser Ausflug in die Dimensionen in Zukunft etwas dabei helfen wird, die - manchmal sehr - verzwickten Zusammenhänge bei mehrdimensionalen Feldern zu verstehen. Bei dem hier durchgespielten Beispiel habe ich die

Dimensionstiefen in Grenzen gehalten, die sich noch relativ leicht nachvollziehen lassen. Stellen Sie sich zum Schluß bitte noch einmal etwas vor:

Ein Gelände mit 100 Lagerhallen. In allen Lagerhallen stehen 60 Regal-Blöcke zu je 50 Regalen. Jedes Regal ist vertikal in 200 Spalten unterteilt und jede dieser Spalten in 40 Regalböden.

Und zu guter Letzt stehen auf jedem Regalboden noch 20 Kästen, wovon wiederum jeder in 10 Fächer aufgeteilt ist.

```
DIM Lager(100,60,50,200,40,20,10)
```

Die sich daraus ergebende Elementeanzahl ist zwar auf einem Amiga nicht zu verarbeiten (100*60*50*200*40*20*10 = 480 Milliarden), aber sie gibt einen Ausblick auf die fast unbegrenzten Einsatzmöglichkeiten von mehrdimensionalen Feldern.

DIM?() — Menge der Feldelemente ermitteln

```
Var=DIM?(Feld())
```

'Feld' ist ein beliebiger numerischer oder alphanumerischer Feldname. DIM? liefert die Anzahl aller Elemente dieses Feldes. Bei nicht dimensionierten Feldern wird der Wert 0 geliefert. Beispiel:

```
Dim Feld%(2,3,4)
If Dim?(Feld%())>0
   Print "Das Feld hat ";Dim?(Feld%());" Elemente"
Else
   Print "Das Feld ist nicht dimensioniert!"
Endif
```

ERASE { ERA } — Feld(er) löschen

```
ERASE Feld()
ERASE Feld1() [,Feld2() [,...]]
```

'Feld' bezeichnet ein beliebiges Feld, das gelöscht werden soll. Die Dimensionierung wird aufgehoben und der dafür reservierte Speicherplatz wieder freigegeben. Außerdem ist es auch möglich, eine Liste von Feldern anzugeben, die dann mit nur einem Befehl gelöscht werden.

Felder, die nach ihrer Verwendung nicht mehr benötigt werden, sollten mit diesem Befehl sofort wieder gelöscht werden, um den durch sie belegten Speicherplatz wieder dem Programm zur Verfügung zu stellen.

Eine wichtige Einsatzmöglichkeit besteht darin, Felder in allgemein verwendbaren Prozeduren (Utilities) einzusetzen. Da nicht erwartet werden kann, daß für jedes Utility die Dimensionierungen im Hauptprogramm vorgenommen werden, sollten diese also in der Prozedur selbst erfolgen. Nach Abschluß der Arbeiten in dieser Prozedur wird das Feld wieder gelöscht, um beim nächsten Aufruf wieder neu dimensioniert werden zu können.

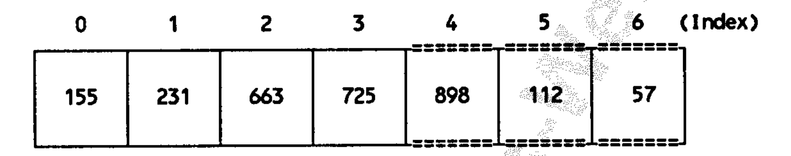
INSERT { INS } — Einzelelement in Feld einfügen

```
INSERT Feld(Index)=Wert
INSERT Feld$(Index)=Text
```

Fügt das einzelne Element Index in das Feld 'Feld' bzw. 'Feld$' mit dem zugewiesenen Wert bzw. String ein. Alle darüberliegenden Elemente werden um eine Stelle nach oben versetzt. Das letzte Element des Feldes wird dabei aus dem Feld entfernt.

Beispiel:

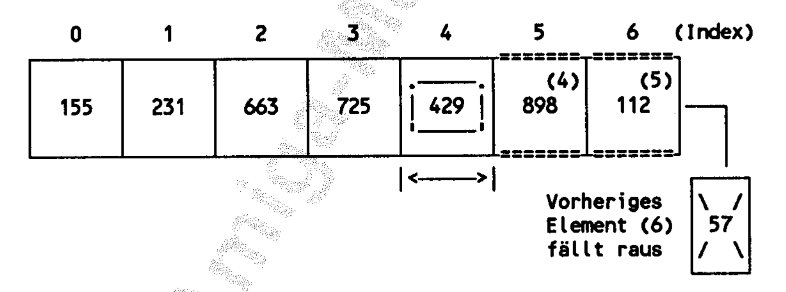
Vorher:



Dann:

INSERT Feld(4)=429

Nachher:



OPTION BASE { OPT BASE } Feld-Basiselement bestimmen

```
OPTION BASE Basis
```

Basis bestimmt das Basis-Element aller Felder (0 oder 1). Die Basis kann im Programm mehrmals geändert werden, da sich die schon definierten Elemente dem neuen Index anpassen (z.B. OPTION BASE 1 -> A$(0) wird A$(1), A$(1) wird A$(2) etc.).

Wurde ein Feld z.B. unter OPTION BASE 0 mit Dim Feld(5) eingerichtet, so ergibt Print Feld(6) eine Fehlermeldung, und das Element Feld(0) ist ohne weiteres ansprechbar. Wird jedoch anschließend OPTION BASE 1 verfügt, so ist das vorher letzte

Element Feld(5) nun ohne Fehlermeldung mit Feld(6) ansprechbar, und das vorher erste Element Feld(0) ergibt eine Fehlermeldung.

Die Basis-Bestimmung wirkt sich auf alle dimensionierten Felder gleichzeitig aus.

QSORT { QS } Feld (-Bereich) Quick-Sortierung

```
QSORT Feld([Sign]) [,Anz [,Feld2%()]]
QSORT Feld$([Sign]) [ WITH Vorgabe()] [,Anz [,Feld2%()]]
```

Es können Felder nach ihrer numerischen Größe oder alphabetischen Reihenfolge sortiert werden. Feld() ist dabei ein numerisches Feld beliebigen Typs. Feld$() ist ein String-Feld.

Das optionale "Sign" (innerhalb der Leerklammer, z.B. QSORT Feld(+)) ist entweder ein Plus- oder Minuszeichen, das die Sortierrichtug angibt. Sollen die Werte bzw. Strings mit dem höchsten Wert bzw. Buchstaben in Element 0 beginnend absteigend sortiert werden, ist ein Minuszeichen einzusetzen. Das Pluszeichen oder keine Sign-Angabe bewirkt die aufsteigende Sortierung.

Anz kann optional verwendet werden, um eine Elementanzahl zu bestimmen, bis zu der sortiert werden soll (z.B. 6 = von 0 - 5 bei OPTION BASE 0 bzw. von 1 - 6 bei OPTION BASE 1). Außerdem kann optional ein 4-Byte-Integerfeld (Feld2%()) angegeben werden, dessen Elemente unabhängig von ihrem Inhalt parallel mit dem eigentlichen Sortierfeld mitsortiert werden.

Bei String-Feldern kann durch WITH zusätzlich ein beliebiges Integerfeld (Vorgabe() => 1-, 2- oder 4-Byte-Integer) mit mindestens 256 Elementen bestimmt werden, dessen Elemente-Inhalte die Reihenfolge der Sortierung vorgibt. Wären z.B. alle 256 Vorgabe()-Elemente mit den ASCII-Werten in normaler Reihenfolge belegt (0-255), so kann Vorgabe() vernachlässigt wer-

den. Werden dagegen z.B. die Zeichen a und A vertauscht, so steht A in der Sortierfolge über a (normalerweise umgekehrt), während alle anderen Zeichen normal sortiert werden. So kann eine völlig willkürliche Sortierfolge vorgegeben werden (siehe Anhang "ASCII-Tabelle").

Beispiel 1:

```
Dim Feld%(20)             ! DIM Feld
Print "unsortiert:"
For I%=0 To 20            ! 20mal
  Feld%(I%)=Rand(100)     ! Zufällige Werte zuweisen
  Print Feld%(I%)         ! Und ausgeben
Next I%                   ! Nächstes Element
Qsort Feld%()             ! Feld sortieren
Print At(15,1);"steigend sortiert:"
For I%=0 To 20            ! 20 sortierte Elemente
  Print At(15,I%+2);Feld%(I%) ! Neu ausgeben
Next I%
```

Beispiel 2:

```
Dim Feld%(20)             ! DIM Feld
Print "unsortiert:"
For I%=0 To 20            ! 20mal
  Feld%(I%)=Rand(100)     ! Zufällige Werte zuweisen
  Print Feld%(I%)         ! Und ausgeben
Next I%                   ! Nächstes Element
Qsort Feld%(-),5          ! 5 Elemente "fallend" sortieren
Print At(15,1);"fallend sortiert:"
For I%=0 To 20            ! 20 Elemente
  Print At(15,I%+2);Feld%(I%) ! Neu ausgeben
  If I%<5                 ! Im Sort-Bereich?
    Print At(20,I%+2);"<-- sortiert"
  Else
    Print At(4,I%+2);"--------->"
  Endif
Next I%
```

Beispiel 3:

```
Dim Feld%(20),Feld2%(20) ! DIM Feld und Parallelfeld
Print "unsortiert:      Indexfeld:"
Print Spc(27);"¦"
For I%=0 To 20            ! 20mal
  Feld%(I%)=Rand(100)     ! Zufällige Werte zuweisen
  Feld2%(I%)=I%           ! Parallelfeld indizieren
  Print Feld%(I%),Feld2%(I%) ! Und ausgeben
Next I%                   ! Nächstes Element
```

```
Qsort Feld%(+),15,Feld2%() ! 15 Elemente + Index sortieren
A$="¦ steigend sortiert:    Indexfeld (mitsortiert):"
Print At(28,1);A$
For I%=0 To 20              ! 20 Elemente
  Print At(28,I%+3);"¦ ";Feld%(I%) ! Feld neu ausgeben
  Print At(52,I%+3);Feld2%(I%) ! Index neu ausgeben
  If I%<15                  ! Im Sort-Bereich?
    Print At(34,I%+3);" -- sortiert --"
  Else
    Print At(34,I%+3);" - unsortiert -"
  Endif
Next I%
```

Beispiel 4:

In den ersten drei Beispielen können Sie statt des 4-Byte-Integerfeldes Feld%() ebensogut andere numerische Feldtypen oder ein String-Feld einsetzen. Der Sortiermodus und das Ergebnis würden dabei prinzipiell gleich bleiben, nur daß bei String-Feldern nicht nach numerischer Reihenfolge, sondern nach der Reihenfolge der den Zeichen entsprechenden ASCII-Werten sortiert würde. Das folgende Beispiel demonstriert das Sortieren nach einer beliebigen Reihenfolge.

```
Dim Feld$(40),Feld2%(40),Vorgabe|(256) ! DIMs
Print "unsortiert:     Indexfeld:"
Print Spc(27);"¦"
X$="äÄöÖüÜß"                    ! Umlaute und Eszet
X2$="AAOOUUS"                   ! Ersatzkriterium
For I%=0 To 255                 ! Alle ASCIIs
  Vorgabe|(I%)=I%               ! In Sortiervorgabe einsetzen
  If Instr("abcdefghijklmnopqrstuvwxyz",Chr$(I%))
    '                           ! Kleinbuchstaben?
    Vorgabe|(I%)=Asc(Chr$(I%-32)) ! Durch Großbuchstaben
    '                           ! Ersetzen
  Endif
  If Instr(X$,Chr$(I%))         ! Umlaut oder Eszet?
    Vorgabe|(I%)=Asc(Mid$(X2$,Instr(X$,Chr$(I%)),1))
    '                           ! Durch Ersatzkriterium
  Endif                         ! Ersetzen
Next I%
For I%=0 To 40                  ! 40mal
  For J%=0 To 8                 ! 8 Zeichen je String
    X%=Rand(59)+65              ! Zufalls-ASCII
    '         A-Z = 26 Zeichen
    '         a-z = 26 Zeichen
    '     ÄÖÜäöüß =  7 Zeichen
    '               ----------
    '   insgesamt = 59 Zeichen.
```

```
      '
      ' Rand(59) wählt also einen Wert zwischen 0 und 58.
      ' Dieser Wert wird anschließend um 65 erhöht
      ' -> ASCII-Wert von 'A' ist 65.
      If X%>115                    ! X% größer A-Z und a-z?
        X%=Asc(Mid$(X$,X%-116,1))! Dann Umlaut oder Eszet
      Else if X%>90                ! X% zwischen a-z?
        Add X%,6                   ! X%+6 = mindesten Asc("a")
      Endif
      Feld$(I%)=Feld$(I%)+Chr$(X%) ! Zeichen einbinden
    Next J%
    Feld2%(I%)=I%                  ! Parallelfeld indizieren
    Print Feld$(I%),Feld2%(I%)     ! Und ausgeben
  Next I%                          ! Nächstes Element
  Qsort Feld$() With Vorgabe|(),41,Feld2%()
  ' Alle Elemente + Index nach Vorgabe steigend sortieren
  A$="| steigend sortiert:  Indexfeld (mitsort.):"
  Print At(28,1);A$
  For I%=0 To 40                   ! 20 Elemente
    Print At(28,I%+3);"| ";Feld$(I%) ! Feld neu ausgeben
    Print At(50,I%+3);Feld2%(I%) ! Index neu ausgeben
  Next I%
```

SSORT { SS } — Feld (-Bereich) Shell-Sortierung

```
SSORT Feld([Sign]) [,Anz [,Feld2%()]]
SSORT Feld$([Sign]) [ WITH Vorgabe()] [,Anz [,Feld2%()]]
```

Erläuterungen zu QSORT gelten hier analog (siehe dort). Der wesentliche Unterschied zwischen beiden Sortier-Algorithmen ist, daß QSORT rekursiv arbeitet und dadurch wesentlich mehr Speicherplatz benötigt als das SSORT-Verfahren. Dafür ist SSORT in den meisten Fällen wesentlich langsamer.

Das Shellsort-Verfahren (es wurde von einem Engländer namens Shell entwickelt) ist aufgrund seines Aufbaus eher dazu geeignet, schon vorsortierte Felder zu sortieren, was beim Quicksort-Verfahren unter Umständen sogar eine Zeitverzögerung bewirken kann. In den meisten Fällen hängt es jedoch von Ihrem persönlichen "Geschmack", dem noch verfügbaren Speicher und der jeweiligen Situation ab, welcher Sortierbefehl für Sie in Frage kommt.

11.2 Speicheroperationen

ABSOLUTE { AB } — Variable auf Adresse setzen

```
ABSOLUTE Var,Adresse
```

Die Adresse der numerischen Variablen Var (beliebiger Typ) wird auf die absolute Speicheradresse Adresse gelegt.

Adresse ist in jedem Fall identisch mit VARPTR(Var):

```
ABSOLUTE Var%,9952
PRINT 9952=V:Var%
      -> Ausgabe = -1 (TRUE)
```

Eine Zuweisung zu einer ABSOLUTE-Variablen ist gleichbedeutend mit:

```
POKE Adresse,Bytewert   -> Bei Byte-Variablen Var|
DPOKE Adresse,Wordwert  -> Bei Word-Variablen Var&
LPOKE Adresse,Longwert  -> Bei Long-Variablen Var%
```

Die Abfrage einer ABSOLUTE-Variablen ist gleichbedeutend mit:

```
Bytewert=PEEK(Adresse)  -> Bei Byte-Variablen Var|
Wordwert=DPEEK(Adresse) -> Bei Word-Variablen Var&
Longwert=DPEEK(Adresse) -> Bei Long-Variablen Var%
```

Bei Real- oder Boole-Variablen ist das jeweilige Zahlenformat zu beachten. ABSOLUTE mit Feld- und String-Variablen ist nicht möglich. ABSOLUTE ist innerhalb von PROCEDUREs und FUNCTIONs mit zuvor als lokal definierten Variablen ebenso möglich wie auf globaler Ebene. Nach Rückkehr zum Hauptprogramm wird die "Absolutierung" jedoch wieder aufgehoben.

Statt des Kommas zwischen Var und Adresse kann auch ein Gleich-Zeichen (ABSOLUTE Var%=Adresse) verwendet werden.

BMOVE { B } Speicherblock kopieren

```
BMOVE Quelle,Ziel,Anz
```

Ab der Adresse Quelle werden Anz Bytes gelesen und an den mit der Adresse Ziel beginnenden Bereich kopiert. Quell- und Zielbereich können sich dabei auch überschneiden.

Bei Anz ist zu beachten, daß nie der Wert 0 auftreten darf. Wird als Anzahl der zu kopierenden Bytes keine Konstante verwendet (z.B. ...,X_bytes*Zeilen), so kann es unter ungünstigen Umständen dazu kommen, daß als Anz-Parameter Null übergeben wird. In solchen Fällen ist meistens ein gnadenloser Absturz die Folge. Um dies zu vermeiden, bietet sich die Funktion MAX(1,Anz) an, die dafür sorgt, daß mindestens ein Byte übertragen wird.

BMOVE arbeitet bei geraden Adressen schneller als bei ungeraden. BMOVE ist ein Befehl, ohne den professionelles Programmieren fast undenkbar wäre. Bei älteren BASIC-Dialekten bestand ein wesentlicher Unterschied zu den maschinennahen Sprachen bzw. Maschinensprache darin, daß kein schneller Speicherblock-Transfer möglich war. Speicherblöcke mußten durch FOR..NEXT-Schleifen und PEEK/POKE Byte für Byte übertragen werden. BMOVE macht dem ein Ende. Das Einsatzgebiet ist so weit gefächert, daß es der Phantasie des Einzelnen überlassen bleibt, was er mit diesem mächtigen Befehl anfängt. An Ideen dürfte es da allerdings nicht mangeln.

BYTE{}, CARD{}, LONG{}

```
Speicherinh. lesen BYTE{}= 1 Byte schreiben CARD{}= 2 Byte schreiben
LONG{}= 4 Byte schreiben
```

```
Syntax (Zuweisung an Adresse/vgl. D-L-POKE):
BYTE{Adresse}=Wert => Schreibt ein Byte
CARD{gerade Adresse}=Wert => Schreibt zwei Byte (Word)
```

```
LONG{gerade Adresse}=Wert => Schreibt vier Byte (Long)
Syntax (Lesen aus Adresse/vgl. D-L-PEEK):
Var=BYTE{Adresse} => Liest ein Byte
Var=CARD{gerade Adresse} => Liest zwei Byte (Word/(Cardinal)
Var=LONG{gerade Adresse} => Liest vier Byte (Long)
```

Ab Adresse werden dem Format entsprechend viele (1, 2, 4) Bytes gelesen bzw. geschrieben. Bei CARD{} und LONG{} dürfen nur gerade Adressen verwendet werden.

Sollen 4-Byte-Werte gelesen werden, kann die Bezeichnung LONG weggelassen werden (z.B. {123456} liest 4 Byte ab Adresse 123456).

CHAR{} — C-Text lesen CHAR{}= ... schreiben

```
Var$=CHAR{Adresse} => C-Text lesen (Funktion)
CHAR{Adresse}=Expr$ => C-Text schreiben
```

Es wird ein String im C-Format gelesen bzw. geschrieben. Bei der Schreibfunktion wird dem String Expr$ automatisch ein Null-Byte angehängt und der String an Adresse geschrieben. Die Lesefunktion liest ab Adresse, bricht beim ersten gefundenen Null-Byte ab und liefert den bis dahin gelesenen Text zurück.

DOUBLE{}, SINGLE{}

```
IEEE-Double/Single-Realformat lesen DOUBLE{}=, SINGLE{}= ... schreiben {
DO }= }, { SI }= }

Syntax (Lesen aus Adresse):
Realvar=DOUBLE{gerade Adresse} => 8-Byte-IEEE-Realzahl
Realvar=SINGLE{gerade Adresse} => 4-Byte-IEEE-Realzahl
Syntax: (Schreiben an Adresse):
DOUBLE{gerade Adresse}=Wert => 8-Byte-IEEE-Realzahl
SINGLE{gerade Adresse}=Wert => 4-Byte-IEEE-Realzahl
```

Beim Schreiben wird der angegebene 8- bzw. 4-Byte-Wert als Wert im IEEE-Format interpretiert und so an die angegebene Adresse geschrieben. Soll ein Wert gelesen werden, wird intern der Inhalt der auf Adresse folgenden 4 bzw. 8 Byte als Wert im IEEE-Format interpretiert und in Realvar zurückgegeben.

FLOAT{}

8 Byte in GFA-3.0-BASIC-Realformat lesen FLOAT{}= 8 Byte in GFA-3.0-Realformat schreiben

```
Realvar=FLOAT{gerade Adresse} => Lesen (Funktion)
FLOAT{gerade Adresse}=Wert => Schreiben
```

Beim Schreiben wird der angegebene Wert als Wert im 8-Byte-Realformat des V3.0-GFA-BASICs interpretiert und so in die - auf Adresse folgenden - 8 Bytes geschrieben. Soll ein Wert gelesen werden, so wird intern der Inhalt der auf Adresse folgenden 8 Byte als V3.0-Realwert interpretiert und in Realvar zurückgegeben.

INT{} /WORD{}

2 Byte als Vorzeichen-Integer schreiben; lesen INT{}= /WORD{}= ... schreiben

```
Intvar=INT{Adresse} => Lesen (Funktion)
Intvar=WORD{Adresse} => Lesen (Funktion)
INT{Adresse}=Wert => Schreiben (Befehl)
WORD{Adresse}=Wert => Schreiben (Befehl)
```

Bei INT{} und WORD{} handelt es sich um zwei Namen derselben Funktion bzw. desselben Befehls. Beim Schreiben wird der angegebene Wert als Wert im vorzeichenbehafteten 2-Byte-Integerformat interpretiert und so in die - auf Adresse folgen-

den - 2 Bytes geschrieben. Soll ein Wert gelesen werden, wird intern der Inhalt der auf Adresse folgenden 2 Byte als vorzeichenbehafteter 2-Byte-Integerwert interpretiert und in der Aufnahmevariablen Intvar (beliebiger Typ) zurückgegeben.

Vorzeichenbehaftete 2-Byte-Werte können nur im Bereich von -32768 (&X1000000000000000) bis 32767 (&X0111111111111111) liegen. Ist also das höchste Bit (Bit 15) der beiden auf Adresse folgenden 16 Bits gesetzt, so ist der gelieferte Wert negativ. Der sich aus den untersten 15 Bits ergebende positive Normal-Integerwert (Wert And (2^15-1)) wird dann zu -32768 addiert und ergibt so den gelieferten Minuswert. Wird als Intvar eine Byte-Variable (z.B. Var|) angegeben, kann damit nur ein vorzeichenloser Wert von 0 - 255 aufgenommen werden.

PEEK, DPEEK, LPEEK — Speicherinhalt auslesen

```
Var=PEEK(Adresse) => Liest ein Byte
Var=DPEEK(gerade Adresse) => Liest zwei Byte (Word)
Var=LPEEK(gerade Adresse) => Liest vier Byte (Long)
```

Ab Adresse werden dem Format entsprechend viele (1, 2 oder 4) Bytes gelesen. Bei DPEEK() und LPEEK() dürfen nur gerade Adressen verwendet werden.

POKE,DPOKE,LPOKE { PO,DP,LP } Speicherinhalt ändern

```
POKE Adresse,Byte => Schreibt ein Byte
DPOKE gerade Adresse,Word => Schreibt zwei Byte (Word)
LPOKE gerade Adresse,Long => Schreibt vier Byte (Long)
```

Schreibt den angegebenen Wert (Byte, Word, Long) im jeweiligen Format in die ab Adresse folgenden 1, 2 oder 4 Bytes. Bei DPOKE und LPOKE dürfen nur gerade Adressen verwendet werden.

Stellen Sie beim Experimentieren mit den POKEs sicher, daß Sie nicht unbeabsichtigt in wichtige Bereiche des Systems oder des Interpreters hinein'poken'. Auch wenn das nicht immer sofort mit einem Absturz endet, können durch falsche Daten in diesen Bereichen Fehlfunktionen ausgelöst werden, die sich erst nach geraumer Zeit bemerkbar machen. Diese Fehlfunktionen sind im Endeffekt wesentlich gefährlicher als ein Absturz, da dadurch auch beim Laden und Speichern Daten unbemerkt "beschädigt" werden können, während man sich in Sicherheit wähnt.

Lassen Sie also bitte grundsätzlich bei allen Speichermanipulationen äußerste Vorsicht walten! Sollte Ihnen ein POKE mal "verloren gehen", (wenn also seine Wirkung nicht nachvollziehbar ist), ist es in den meisten Fällen angebracht, die Daten (Programm etc.) so schnell wie möglich auf Disk zu sichern und dann einen Reset auszulösen, ehe Schlimmeres folgt.

11.3 Speicherverwaltung

INLINE { INL } — **BASIC-interne Speicherreservierung**

```
INLINE Adresse%,Bytes
```

Reserviert innerhalb des Programm-Listings einen Speicherbereich von maximal 32700 Bytes. Dazu wird in 'Bytes' die gewünschte Größe angegeben. Adresse% ist eine 4-Byte-Integer-Rückgabevariable (keine Feldvariable). Trifft das Programm auf eine INLINE-Zeile, so wird darin vom BASIC die aktuelle Startadresse des INLINE-Speichers geliefert. Sonst geschieht nichts. Ob etwas im Speicher ist, bzw. was Sie mit der Adresse und diesem Speicher anfangen, hängt davon ab, was Sie bei der Programmerstellung in diesen Speicher geladen haben.

Die Besonderheit dieses Befehls besteht nämlich darin, daß schon der Editor auf die Eingabe einer INLINE-Zeile reagiert. Überall dort, wo eine INLINE-Zeile geschrieben wurde und

durch <Return> oder eine andere Editorfunktion verlassen wird, wird im Programmspeicher ein Bereich mit der angegebenen Größe reserviert. Wenn Sie nun das Programm mit der Editorfunktion Save bzw. SAVE oder PSAVE abspeichern, werden diese INLINE-Speicher als Programmbestandteil mitsamt Inhalt mit abgespeichert. Beim nächsten Laden durch Load bzw. LOAD werden die INLINE-Speicher ebenfalls mitgeladen, sie sind dann unverändert wieder verfügbar.

Mit der Editorfunktion Save,A bzw. dem Befehl LIST wird nur die Zeile, aber nicht der INLINE-Speicher gesichert. Beim Laden eines ASCII-Listings mit Merge wird zwar wieder der Speicher reserviert, aber er ist dann ohne Inhalt.

In einer INLINE-Befehlszeile ist kein !-Kommentar möglich, da an Stelle des Kommentars intern der INLINE-Speicher gelegt wird. Beim ersten Anlegen eines Speichers wird dieser mit Null-Bytes gefüllt. Wird eine bestehende INLINE-Zeile aus dem Programm gelöscht, wird auch automatisch der Speicher wieder an das BASIC zurückgegeben.

Wird dagegen eine bereits bestehende INLINE-Zeile nachträglich verändert (neuer Variablename oder neue Speichergröße), fragt der Interpreter zuerst nach, ob der alte INLINE-Speicher gelöscht werden soll. Wird diese Abfrage mit "OK" beantwortet, wird der alte Speicher gelöscht und ein neuer - den neuen Angaben entsprechender - Speicher eingerichtet.

Steht der Cursor auf einer fertigen (schon gecheckten) INLINE-Zeile, kann man die <Help>-Taste drücken. Das können Sie sonst natürlich auch, nur hat es dann keinen Zweck. Drücken Sie also die <Help>-Taste, während der Cursor auf einer INLINE-Zeile steht, so erscheint eine Menüzeile.

Mit Mausklick auf Load kann eine beliebige Datei in den reservierten Bereich geladen werden (z.B. Maschinen-Code, Bilddateien etc.). Mit Save wird der INLINE-Speicher auf Diskette gesichert. Dump ermöglicht die Ausgabe des INLINE-Inhalts in

2-Byte-Hexwerten (mit Offset-Angabe) auf dem Drucker, und Clear löscht (ohne Sicherheitsabfrage) den INLINE-Speicher.

Beim Speichern wird der INLINE-Datei - sofern keine andere angegeben wurde - die Extension .INL "verpaßt". Beim Laden wird ebenfalls die Auswahl-Vorgabe .INL voreingestellt. Es kann jedoch jede beliebige Datei geladen werden. Die zu ladende Datei sollte entweder genauso groß wie der INLINE-Speicher oder größer sein. Dateien, die den INLINE-Speicher nicht ganz füllen, werden mit der Meldung "File-Ende erreicht" abgewiesen. Dateien mit einer Länge, die größer als der INLINE-Speicher ist, werden bei der INLINE-Länge abgeschnitten.

Noch einmal das Ganze in Stichpunkten:

1. INLINE-Zeile schreiben (Rückgabe-Variable und Größe angeben); wenn der INLINE-Speicher gefüllt, gesichert, gedruckt oder gelöscht werden soll:

2. Cursor auf INLINE-Zeile und <Help> drücken.

3. Menü-Funktion wählen.

MALLOC() — System-Speicher-Reservierung

```
Back=MALLOC(Anz,Art)
```

MALLOC reserviert 'Anz' Bytes an Systemspeicher. Dieser Speicherbereich ist anschließend gegen den Zugriff von anderen (parallel ablaufenden) Programmen aus geschützt. Als Rückgabewert erhält man bei durchgeführter Reservierung die Startadresse des reservierten Bereichs (im Falle eines Fehlers enthält 'Back' den Wert Null). Diese Adresse sollte man sich unbedingt merken, damit der Speicher später auch wieder (mit MFREE) freigegeben werden kann.

Hinweis: Der Wert in 'Anz' wird von MALLOC immer auf das nächste Vielfache von 8 aufgerundet.

Die Variable 'Art' bestimmt, welcher Art der zugewiesene Speicher sein soll:

Art=2 Der zu reservierende Speicherbereich soll im sog. Chip-Memory liegen. Chip-Memory ist der untere 512-Kilobyte-Bereich des Speichers. Nur dieser Bereich kann von den Spezialchips zur Grafik- und Sounderzeugung angesprochen werden! Zur Speicherung von Grafik- und Sounddaten müssen Sie also unbedingt diese Art von Speicher reservieren lassen.

Art=4 Der zu reservierende Bereich soll sich außerhalb des Chip-Memories im sog. Fast-Memory befinden. Das Ganze hat natürlich nur dann einen Sinn, wenn Sie auch über mehr als 512 Kilobyte Gesamtspeicher verfügen.

Art=1 Der zu reservierende Bereich darf - nachdem er festgelegt wurde - nicht mehr verschoben werden. In der momentanen Version des Betriebssystems ist es zwar noch nicht vorgesehen, einmal reservierten Speicher nachträglich zu verschieben. Aus Kompatibilitätsgründen empfiehlt es sich aber, diese Speicherart trotzdem zu wählen, insbesondere dann, wenn Sie den reservierten Speicher längere Zeit benötigen.

'Art=1' Kann durch ODER-Verknüpfung mit den anderen Speicherarten verknüpft werden. 'Art= 1 OR 2' zum Beispiel reserviert nicht verschiebbares Chip-Memory.

Art=65536 Der reservierte Speicherbereich wird mit Nullen aufgefüllt. Diese Speicherart kann mit allen anderen durch ODER-Verknüpfung kombiniert werden.

Hinweis: Falls Art weder die Kennung für Chip- noch für Fast-Memory enthält (z.B. 'Art=1'; Bereich darf nicht verschoben werden, wo er liegt, ist aber egal), so wird zuerst versucht, Fast-Memory zu belegen. Wenn sich dort kein genügend großer freier Bereich mehr findet, wird Chip-Memory reserviert.

Beispiele:

```
Back=MALLOC(1000,4 OR 65536)
```

Belegt 1000 Byte außerhalb des Chip-Memories und füllt den reservierten Bereich mit Nullen auf.

```
Back=MALLOC(32000,2)
Reserviert 32000 Byte im Chip-Memory
Back=MALLOC(150,2 OR 1 OR 65536)
```

Belegt 150 Byte nicht verschiebbares Chip-Memory und füllt den reservierten Bereich mit Nullen auf.

MFREE() MALLOC()-Speicher freigeben

```
Back=MFREE(Adresse,Anz)
```

Gibt den durch MALLOC() reservierten Speicher wieder frei. In 'Adresse' wird die Startadresse des freizugebenden Bereichs angegeben (bei MALLOC()-Aufruf merken), in 'Anz' muß die Größe des Bereichs übergeben werden. In 'Back' wird die Größe des freigegebenen Bereichs (also Anz) zurückgegeben.

Vorsicht: Bitte achten Sie sorgfältig darauf, daß die an MFREE übergebenen Parameter auch wirklich stimmen! Geben Sie zum Beispiel eine falsche Adresse oder (in Anz) eine falsche Bereichsgröße an, so bekommen Sie unweigerlich einen Systemabsturz (Guru-Meditation).

RESERVE { RESE } — BASIC-Arbeitsspeicher festlegen

```
RESERVE [Anz]
```

'Anz' gibt die neu einzurichtende Größe des BASIC-Arbeitsspeichers (Programm+Variablen) in Bytes an. Voreingestellt sind 64 KByte. Dieser Wert wird auch genommen, wenn man bei RESERVE den Parameter Anz wegläßt.

Achtung: Durch RESERVE wird der Variablenspeicher komplett gelöscht!

11.4 Zeigeroperationen

* — Variablen-Pointer

```
Var=*Var
Var=*Feld()
```

Wird ein Variablenname Var bzw. Feld() beliebigen Typs (siehe TYPE) auf diese Art als Zeiger gekennzeichnet, wird nicht der Variableninhalt, sondern die Variablenadresse übergeben. Bei Feldern und Strings ist dies der zugehörige Descriptor (-> ARRPTR X() oder ARRPTR X$), bei numerischen Variablen die Adresse, an welcher der Variableninhalt zu finden ist.

Beispiel 1:

```
A%=12          ! A% mit 12 belegen
B%=*A%         ! Pointer auf A% in B% speichern
*B%=33         ! Indirekt A% mit 33 belegen
Print "Variablenadresse A% (über Pointer): ";*A%
Print "Variablenadresse A% (über VARPTR) : ";Varptr(A%)
Print "Variablenadresse A% (in B%)       : ";B%
Print "Variableninhalt A%  (direkt)      : ";A%
Print "Variableninhalt A%  (indirekt)    : ";Lpeek(B%)
```

Beispiel 2:

```
Var$="GFA-BASIC"             ! String-Variable belegen
Var%=5                       ! 4-Byte-Integervariable belegen
Gosub Proc(*Var$,*Var%)      ! Proc-Aufruf mit Pointer-Variablens
Print Var$,Var%              ! Globale Variablen ausgeben
Procedure Proc(Para1%,Para2%) ! Kopf mit Pointer-Aufnahme
  Local Lvar$,Lvar,Lvar%,I%    ! Lokale Variablen vorbereiten
  For I%=0 To Dpeek(Para1%+4)-1 ! String-Länge aus Descriptor
    Z%=Peek(Lpeek(Para1%)+I%) ! Zeichen lesen (mittels
    '                        ! des Descriptors durch PEEK)
    Lvar$=Lvar$+Chr$(Z%)     ! Lokalen String bilden
  Next I%                    ! Nächstes Zeichen
  If Type(Para2%)=0          ! Para2%=Realvariable? (siehe TYPE())
    Bmove Para2%,Varptr(Lvar),8 ! 8 Bytes in die lokale
    '                           ! Realvariable übertragen
    *Para1%=String$(3,Lvar$)+" / 2^"+Str$(Lvar)+" = "
    '                        ! String bilden und zurückgeben
    *Para2%=2^Lvar           ! Ergebnis berechnen und zurückgeben
  Endif
  If Type(Para2%)=2
    Lpoke Varptr(Lvar%),Lpeek(Para2%) ! 4 Bytes in die lokale
    '                        ! Integer-Variable übertragen
    *Para1%=String$(3,Lvar$)+" / 2^"+Str$(Lvar%)+" = "
    '                        ! String bilden und zurückgeben
    *Para2%=2^Lvar%          ! Ergebnis berechnen und zurückgeben
  Endif
Return
```

Für die Variablen- und Feldübergabe an Unterprogramme eignet sich dagegen VAR. Hiermit ist es möglich, die Variablen und Felder "direkt" zu übergeben. Sie sind dann gleichzeitig Übergabevariablen und Rückgabevariablen. Eine Übernahme der Variableninhalte wie in Beispiel 2 ist hier nicht mehr nötig.

ARRPTR — String-/Feld-Descriptoradresse ermitteln

```
Var=ARRPTR(Var$)
Var=ARRPTR(Feld())
```

Liefert die Anfangsadresse des String- bzw. Feld-Descriptors (siehe Erläuterungen im Anhang "Variablenorganisation/-typen").

VARPTR — Variablen-Adresse ermitteln { V: }

```
Var=VARPTR(Var)
```

Liefert bei numerischen Variablen deren Adresse bzw. bei String-Variablen die Adresse des ersten Zeichens. Var steht für jede beliebige Variable (auch Feldelement). Es kann auch V: als Abkürzung verwendet werden (z.B. PRINT V:A$).

Beispiel:

```
A$="BASIC"          ! String setzen
Adr%=Varptr(A$)     ! String-Adresse holen
For I%=0 to 5       ! 5 Zeichen
  Print Chr$(Peek(ADR%+I%)); ausgeben
Next I%
```

11.5 Die Exec-Bibliothek des Amiga

Die Exec-Bibliothek beinhaltet in der überwiegenden Mehrzahl Funktionen zur Multitasking-, zur Interrupt- und zur Speicherverwaltung des Amiga. Dinge also, mit denen man sich in GFA-BASIC - zum Glück - nur äußerst selten befassen muß.

Um die Funktionen der Exec-Bibliothek erfolgreich anwenden zu können, sind detaillierte Kenntnisse über Aufbau und Funktionsweise von AmigaDOS erforderlich, die den Rahmen dieses Buches bei weitem sprengen würden. Bitte nehmen Sie es mir also nicht übel, wenn ich im folgenden die einzelnen Funktionen nur in sehr knapper Form vorstelle.

Wer mehr wissen möchte, der sollte einmal einen Blick in 'Amiga Intern', 'Intern 2' oder in das große AmigaDOS-Buch werfen.

Und bitte denken Sie immer daran: Viele der Exec-Routinen beeinflussen die elementarsten Abläufe innerhalb des Amiga!

Wenn da etwas schiefläuft, bekommen Sie in vielen Fällen nicht einmal mehr eine Guru-Meditation.

Für alle, die es dennoch wagen wollen, die Routinen zu nutzen, noch ein Hinweis: Alle Exec-Routinen lassen sich von GFA-BASIC aus jederzeit wie eine GFA-BASIC-Funktion ansprechen. Irgendwelche Öffnungszeremonien, wie man sie von Amiga-BASIC kennt, sind nicht erforderlich.

Die verfügbaren Funktionen im einzelnen:

SetTaskPri	Priorität eines Tasks ändern
Prioralt	SetTaskPri(Task,Priorneu)

Gleich zu Anfang die vielleicht interessanteste Funktion. SetTaskPri ändert die Priorität (-128 bis +127) eines Tasks. (In 'Prioralt' wird die alte Priorität zurückgegeben.)

Dazu einige Erläuterungen: Auch wenn es so scheint, als ob der Amiga in der Lage wäre, mehrere Programme gleichzeitig zu verarbeiten (Stichwort: Multitasking), wird in Wirklichkeit zu einem bestimmten Zeitpunkt immer nur ein Programm abgearbeitet. Der Trick beim Multitasking besteht einfach darin, zwischen den einzelnen Programmen in so schneller Folge umzuschalten, daß für den Anwender der Eindruck entsteht, als würden alle Programme gleichzeitig ablaufen. Jedes Programm läuft dazu in einem sog. Task. Das Umschalten zwischen den einzelnen Tasks besorgt das Betriebssystem.

Jeder Task erhält einen bestimmten Anteil an Rechenzeit. Im einfachsten Fall wird die Rechenzeit zwischen den einzelnen Tasks gleichmäßig aufgeteilt. Sind also zum Beispiel drei Tasks gleichzeitig aktiv, so erhält jeder Task ein Drittel der Rechenzeit zugeteilt. In vielen Fällen möchte man diese lineare Aufteilung der Rechenzeit jedoch vermeiden. Ein Beispiel: Nehmen wir einmal an, Sie arbeiten mit zwei GFA-BASIC-Programmen gleichzeitig. Das eine Programm führt komplexe mathematische Berechnungen durch, während das andere Programm gerade eine Grafik zu Papier bringt. Wenn Sie nicht gerade über einen Laserdrucker verfügen, dürfte das zweite Programm die meiste

Zeit damit beschäftigt sein, auf den Drucker zu warten, bis dieser jeweils seinen Datenpuffer ausgedruckt hat. Hier wäre es günstiger, dem Mathematikprogramm den Löwenanteil an Rechenzeit zuzuteilen.

Zu diesem Zweck besteht die Möglichkeit, jedem Task eine bestimmte Priorität zuzuweisen. Die Skala reicht dabei von -127 bis +128. Je höher die Priorität eines Tasks ist, desto größer ist sein Anteil an Rechenzeit.

Wie geht man nun konkret vor? Das ist im Grunde genommen ganz einfach. Bleiben wir beim Beispiel. Zunächst gehen Sie in das Mathematikprogramm und geben dort im Direktmodus ein:

```
VOID SetTaskPri(FindTask(0),5)
```

Anschließend schalten Sie auf das Druckprogramm um und geben dort - ebenfalls im Direktmodus - ein:

```
VOID SetTaskPri(FindTask(0),-5)
```

Der eine Task erhält also die Priorität 5, der andere die Priorität -5. Diese relativ nahe beieinander liegenden Werte reichen bei nur zwei Programmen völlig aus.

Natürlich können Sie SetTaskPri auch innerhalb eines Programms (und nicht nur im Direktmodus) verwenden. Um wieder auf die Standardpriorität, nämlich 0, zurückzuschalten, gibt es zwei Möglichkeiten:

- VOID SetTaskPri(FindTask(0),0)

- Anklicken des Pulldown-Menüpunktes 'TaskPri 0' im GFA-Editormenü.

Supervisor – In Supervisor-Modus umschalten

```
VOID Supervisor()
```

Schaltet den 68000er-Prozessor des Amiga in den Supervisor-Modus.

InitCode – Resident-Module initialisieren

```
VOID InitCode(Startwert, Versionsnummer)
```

Initialisiert alle Resident-Module mit dem angegebenen Startwert und der Versionsnummer.

InitStruct – Speicherbereich initialisieren

```
VOID InitStruct(Init, Puffer, Größe)
```

Initialisiert den Speicherbereich der Länge 'Größe' beginnend ab der Adresse 'Puffer' mit den Werten, die ab Adresse 'Init' im Speicher stehen.

MakeLibrary – Neue Funktionsbibliothek erzeugen

```
Adr = MakeLibrary(...)
```

Erzeugt eine neue Funktionsbibliothek und übergibt deren Adresse in 'Adr'.

FindResident – Adresse einer Resident-Struktur suchen

```
Adr = FindResident(Name)
```

Versucht die Adresse der Resident-Struktur 'Name' zu finden und übergibt diese gegebenenfalls in 'Adr'.

InitResident – Resident-Struktur initialisieren

```
VOID InitResident(Adr, Seglist)
```

Initialisiert eine Resident-Struktur (Adresse in 'Adr') unter Benutzung der Segmentliste, deren Adresse in 'Seglist' steht.

Debug ______ **Debug-Routine aktivieren**

```
VOID Debug()
```

Aktiviert die ROM-Wack-Funktion des Betriebssystems. Danach werden die Debug-Daten (zur externen Kontrolle des ablaufenden Programms durch andere Rechner) über die RS-232-Schnittstelle geschickt.

Disable ______ **Interrupts sperren**

```
VOID Disable()
```

Sperrt diverse Interrupts.

Enable ______ **Interrupts wieder aktivieren**

```
VOID Enable()
```

Aktiviert die durch die Funktion Disable desaktivierten Interrupts wieder.

Forbid ______ **Task-Switching abschalten**

```
VOID Forbid()
```

Desaktiviert das Multitasking-System des Amiga.

Permit ______ **Task-Switching wieder einschalten**

```
VOID Permit()
```

Gegenstück zu Forbid. Reaktiviert das Multitasking.

SetSr ______ **Statusregister verändern**

```
Alt = SetSr(Neu, Maske)
```

Verändert das Statusregister des 68000er-Prozessors. In 'Alt' wird der alte Wert des Registers zurückgegeben.

SuperState — In Supervisor-Modus umschalten

```
Stack = SuperState()
```

Schaltet den 68000er-Prozessor in den Supervisor-Modus. Stack enthält den alten Stackpointer. Dieser Wert muß (!) für den Aufruf von UserState aufbewahrt werden.

UserState — In User-Modus umschalten

```
VOID UserState(Stack)
```

Schaltet den 68000er-Prozessor in den User-Modus. 'Stack' enthält den Wert, den Sie bei SuperState zurückerhalten haben.

SetIntVector — System-Interrupt-Vektor setzen

```
Adralt = SetIntVector(Nummer, Adrneu)
```

Setzt den Interrupt-Vektor neu und übergibt in 'Adralt' die Adresse der alten Struktur. (Adresse der neuen Struktur befindet sich in 'Adrneu'.)

AddIntServer — Interrupt-Server-Routine einfügen

```
VOID AddIntServer(Nummer, Adr)
```

Fügt zu den bereits vorhandenen Interrupt-Server-Routinen eine neue hinzu.

Cause — Software-Interrupt ausführen

```
VOID Cause(Adr)
```

Führt einen Software-Interrupt aus, dessen Adresse in 'Adr' stehen muß.

Allocate ______________________ Speicherblock anfordern

```
Adr = Allocate(Memheader, Größe)
```

Fordert aus einer eigenen Speicherlistenverwaltung einen Block an. Im Erfolgsfall enthält 'Adr' die Adresse des Blockanfangs.

Deallocate ______________________ Speicherblock freigeben

```
VOID Deallocate(Memheader, Adr, Größe)
```

Gibt einen mit der Funktion Allocate angeforderten Speicherblock wieder frei.

AllocMem ______________________ Speicherplatz reservieren

```
Adr = AllocMem(Größe, Art)
```

Diese Funktion entspricht der GFA-BASIC-Funktion MALLOC.

FreeMem ______________________ Speicherplatz freigeben

```
Größe = FreeMem(Adr, Größe)
```

Diese Funktion entspricht der GFA-BASIC-Funktion MFREE.

AvailMem ______________________ Freien Speicherplatz ermitteln

```
Speicher = AvailMem(Art)
```

Liefert den im Moment verfügbaren freien Speicherplatz. 'Art' enthält die Speicherart (Chip-Memory usw.).

AllocEntry ______________________ MemList-Struktur initialisieren

```
Adr = AllocEntry(MemList)
```

Initialisiert eine MemList-Struktur und übergibt im Erfolgsfall deren Adresse in 'Adr'.

FreeEntry ______ **Speicherbereiche löschen**

```
VOID FreeEntry(MemList)
```

Löscht alle Speicherbereiche, die durch die Funktion AllocEntry in einer MemList-Struktur reserviert wurden.

Insert ______ **Node-Struktur einfügen**

```
VOID Insert(List, Node1, Node2)
```

Fügt eine Node-Struktur in eine Node-Strukturen-Liste ein.

AddHead ______ **Node-Struktur am Anfang einfügen**

```
VOID AddHead(List,Node)
```

Fügt eine Node-Struktur an den Anfang einer doppelt verketteten Liste ein.

AddTail ______ **Node-Struktur am Ende einfügen**

```
VOID AddTail(List,Node)
```

Fügt eine Node-Struktur an das Ende einer doppelt verketteten Liste ein.

Remove ______ **Node-Struktur löschen**

```
VOID Remove(Node)
```

Löscht eine Node-Struktur aus einer verketteten Liste von Node-Strukturen.

RemHead ______ **Node-Struktur am Anfang löschen**

```
Adr= RemHead(List)
```

Löscht das erste Element aus einer verketteten Liste von Node-Strukturen. 'List' enthält die Adresse der verketteten Liste, in 'Adr' wird die Adresse der gelöschten Node-Struktur zurückgegeben.

RemTail ______ Node-Struktur am Ende löschen

```
Adr = RemTail(List)
```

Löscht das letzte Element aus einer verketteten Liste von Node-Strukturen. 'List' enthält die Adresse der verketteten Liste, in 'Adr' wird die Adresse der gelöschten Node-Struktur zurückgegeben.

Enqueue ______ Node-Struktur in Systemliste einfügen

```
VOID Enqueue(List, Node)
```

FindName ______ Node-Struktur nach Namen durchsuchen

```
Adr = FindName(Start,Name)
```

Durchsucht eine System-Node-Liste (Adresse in 'Start') nach dem in 'Name' angegebenen Namen und übergibt gegebenenfalls in 'Adr' die Adresse der zugehörigen Node-Struktur.

AddTask ______ Neuen Task anmelden

```
VOID AddTask(TaskCB,InitialPC,FinalPC)
```

Meldet dem System einen neuen Task an.

RemTask ______ Task beenden

```
VOID RemTask(Task)
```

Beendet den angegebenen Task und löscht ihn.

FindTask ______ Task suchen

```
Adr = FindTask(Name)
```

Durchsucht die Task-Listen nach dem Task mit dem angegebenen Namen und übergibt gegebenenfalls dessen Adresse in 'Adr'.

SetSignal — Signalbit-Status setzen

```
Sigalt = SetSignal(Signeu, Sigmaske)
```

Setzt den Status der Task-Empfangs-Signalbits. 'Signeu' enthält den neuen Status, in 'Sigalt' wird der alte Status zurückgegeben.

SetExcept — Signalbit auswählen

```
Sigalt = SetExcept(Signeu, Maske)
```

Wählt eines der insgesamt 32 Signalbits eines Tasks aus. In Sigalt wird das alte Signalbit zurückgegeben.

Wait — Auf Signal warten

```
VOID Wait(Signal)
```

Hält einen Task solange an, bis (von einem anderen Task) ein Signal gesendet wird.

Signal — Signal senden

```
VOID Signal(Task, Signal)
```

Schickt ein Signal an einen anderen Task.

AllocSignal — Signalbit reservieren

```
Rück = AllocSignal(Signalnr)
```

Reserviert eines der verfügbaren Signalbits.

FreeSignal — Signalbit freigeben

```
VOID FreeSignal(Signalnr)
```

Gibt ein mit AllocSignal reserviertes Signalbit wieder frei.

AllocTrap ______________________ Trap-Nummer reservieren

```
Trapnr = AllocTrap(Trapnr)
```

Reserviert eine Trap-Nummer der verfügbaren Traps der laufenden Task.

FreeTrap ______________________ Trap-Nummer freigeben

```
VOID FreeTrap(Trapnr)
```

Gibt einen mit der Funktion AllocTrap reservierten Trap wieder frei.

AddPort ______________________ Message-Port anfügen

```
VOID AddPort(MsgPort)
```

Fügt an die System-Message-Port-Liste einen neuen Message-Port und macht ihn jedem Task zugänglich.

RemPort ______________________ Message-Port löschen

```
VOID RemPort(MsgPort)
```

Löscht einen Message-Port aus der Message-Port-Liste.

PutMsg ______________________ Nachricht anhängen

```
VOID PutMsg(MsgPort, Message)
```

Hängt eine Nachricht an den angegebenen Message-Port.

GetMsg ______________________ Nachricht holen

```
MsgPort = GetMsg(MsgPort)
```

Holt die nächste Nachricht aus dem angegebenen Message-Port.

ReplyMsg ______ **Nachricht zurücksenden**

```
VOID ReplyMsg(Message)
```

Sendet eine empfangene Nachricht an den absendenden Message-Port zurück (zur Empfangsbestätigung).

WaitPort ______ **Auf Ereignis warten**

```
VOID WaitPort(MsgPort)
```

Wartet auf ein Ereignis in dem angegebenen Message-Port.

FindPort ______ **Message-Port suchen**

```
Adr = FindPort(Name)
```

Durchsucht die System-Message-Port-Liste nach einem Port mit dem angegebenen Namen und übergibt gegebenenfalls in 'Adr' dessen Adresse.

AddLibrary ______ **Neue Library einfügen**

```
VOID AddLibrary(Adr)
```

Fügt eine neue Library zum System hinzu und macht sie für alle Tasks zugänglich.

RemLibrary ______ **Library entfernen**

```
Rück = RemLibrary(Adr)
```

Löscht eine Library aus der Library-Liste des Systems.

CloseLibrary ______ **Library schließen**

```
VOID CloseLibrary(Adr)
```

Schließt eine Library, deren Adresse sich in 'Adr' befindet.

SetFunction — Neue Funktion in Library einfügen

```
Adr = SetFunction(Library, Funkoffset,Funkadr)
```

Fügt in eine Funktionsbibliothek eine neue Funktion ein. In 'Adr' wird die Adresse der alten Funktion zurückgegeben, die an der angegebenen Position in der Bibliothek stand.

SumLibrary — Neue Checksumme berechnen

```
VOID  SumLibrary(Library)
```

Berechnet für die angegebene Library eine neue Checksumme.

AddDevice — Device hinzufügen

```
VOID AddDevice(Adr)
```

Fügt zur System-Device-Liste ein neues Device hinzu und macht es für alle Tasks zugänglich.

RemDevice — Device entfernen

```
VOID RemDevice(Adr)
```

Löscht ein Device aus der System-Device-Liste.

OpenDevice — Device öffnen

```
Rück = OpenDevice(Name, Nummer, IORequest, Flags)
```

Öffnet das durch seinen Namen bezeichnete Device und initialisiert die zugehörige IORequest-Struktur.

CloseDevice — Device abmelden

```
VOID CloseDevice(IORequest)
```

Meldet ein Device beim System ab.

DoIO ______ **Kommando ausführen**

```
Rück = DoIO(IORequest)
```

Führt das in der IORequest-Struktur festgelegte Kommando aus.

WaitIO ______ **Auf Kommandoausführung warten**

```
Rück = WaitIO(IORequest)
```

Wartet, bis das in der IORequest-Struktur angegebene Kommando ausgeführt ist.

AddResource ______ **Resource hinzufügen**

```
VOID AddResource(Adr)
```

Fügt ein Resource zur System-Resource-Liste hinzu und macht es jedem Task verfügbar.

RemResource ______ **Resource entfernen**

```
VOID RemResource(Adr)
```

Entfernt ein Resource aus der System-Resource-Liste.

OpenResource ______ **Resource öffnen**

```
Adr = OpenResource(Name, Version)
```

Öffnet ein bereits installiertes Resource und übergibt in 'Adr' die Adresse der zugehörigen Resource-Struktur.

OpenLibrary ______ **Library öffnen**

```
Adr = OpenLibrary(Name, Version)
```

Öffnet eine Funktionsbibliothek und übergibt in 'Adr' die Adresse der zugehörigen Librarystruktur.

12. Programmkontrolle

12.1 Programmstart und -ende

CONT { CON } — **Programm (nach STOP-Befehl) fortsetzen**

```
CONT
```

Wurde der Programmlauf mit STOP unterbrochen, kann durch CONT im Direktmodus das Programm in der Zeile nach dem STOP-Befehl fortgesetzt werden. CONT ist nicht möglich, wenn nach STOP entweder CLEAR verwendet, das Programm-Listing verändert oder neue Variablen eingeführt wurden.

In manchen Fällen kann es notwendig sein, in ON BREAK GOSUB- oder ON ERROR GOSUB-Prozeduren CONT einzusetzen. Ob ein solcher Fall eintritt, hängt von verschiedenen Umständen ab, die sich nicht immer konkret vorhersagen lassen. Wenn es Ihnen passiert, daß das Programm in solchen Prozeduren "hängenbleibt", ist es gut, daß Sie von dieser Möglichkeit schon einmal gehört haben.

EDIT { ED } — **Programm beenden**

```
EDIT
```

Hat dieselbe Wirkung wie END (siehe dort). EDIT kehrt jedoch ohne Vorwarnung direkt zum Editor (Interpreter) bzw. zur Workbench (bei Kompilaten) zurück.

END — Programm beenden

```
END
```

Bewirkt den Abbruch des aktuellen Programms. Variableninhalte/offene Dateien bleiben im Interpreter bis zur nächsten Programmänderung bzw. bis zum nächsten CLEAR erhalten/geöffnet und können im Direktmodus weiter angesprochen werden.

Das Programm kann danach nicht durch CONT fortgesetzt werden. Im Interpreter-Betrieb erscheint eine Programm-Ende-Meldung, nach der zum Editor zurückgekehrt wird. Kompilate kehren ohne Ende-Meldung direkt zur Workbench bzw. zum aufrufenden Programm (siehe EXEC) zurück.

QUIT { Q } — Programmende (Rückkehr zu CLI od. Workbench)

```
QUIT
QUIT [x] (nur V3.0)
```

QUIT ist identisch mit SYSTEM und bewirkt, daß das Programm ohne jegliche Sicherheitsabfrage zur Workbench bzw. (evtl. bei Kompilaten) zum Aufrufer (also dem CLI oder EXEC) zurückkehrt.

In Version V3.0 kann in x ein 16-Bit-Wert angegeben werden, der an das aufrufende Programm (über D0) zurückgegeben und dann dort ausgewertet werden kann (siehe EXEC als Funktion).

Allgemeine Konvention:

x=0	Programm wurde korrekt - ohne Error - verlassen.
x>0	BASIC-Fehler aufgetreten (ERR = 0 bis 109).
x<0	System-Fehler aufgetreten (ERR= -1 bis -67).

Tritt ein System- oder BASIC-Fehler ein, könnte ggf. QUIT ERR in einer Fehler-Abfangroutine (siehe ON ERROR GOSUB) als Programmende eingesetzt werden. So "weiß" ggf. das aufrufende Programm, aus welchen Gründen das von ihm aufgerufene Programm beendet wurde.

RUN { RU } — Programm starten

```
RUN
RUN "Programmname"
```

Startet das aktuelle Programm neu. Dabei werden sämtliche Variablen und der Bildschirm gelöscht. RUN kann auch im Direktmodus verwendet werden.

In der uns vorliegenden Version V3.0 kann ein Programmname angegeben werden. Das angegebene Programm wird geladen und automatisch gestartet (vgl. CHAIN).

STOP { ST } — Programm unterbrechen

```
STOP
```

Mit STOP kann der Programmlauf an jeder Stelle unterbrochen werden. Es erscheint eine ALERT-Box, durch die man das Programm fortsetzen kann oder durch die man in den Direktmodus gelangt.

Da keine Variablen gelöscht und keine Dateien geschlossen werden, kann aus dem Direktmodus heraus durch Eingabe einzelner Befehlszeilen schrittweise gearbeitet werden und anschließend das Programm ggf. durch Eingabe von CONT im Direktmodus fortgesetzt werden (siehe CONT). Es kann auch zum Editor gewechselt werden. Solange dort keine Zeilen verändert werden,

kann auch dann nach Rückkehr zur Direkt-Eingabeebene durch CONT das Programm wieder aufgenommen werden.

Beispiel:

```
DO                  ! Endlos-Schleife
  INC Aa            ! Irgendeine Zählvariable +1
  IF Aa>100         ! Variable größer 100?
    CLR Aa          ! Variable löschen
    STOP            ! Programmstop
  ENDIF
  PRINT Aa''        ! Wert ausgeben
LOOP
```

SYSTEM { SYS } — Programmende (Interpreter verlassen)

```
SYSTEM
SYSTEM [x]
```

Ist identisch mit QUIT (siehe Erläuterungen dort).

12.2 Löschfunktionen

CLEAR { CLE } — Felder und Variablen löschen

```
CLEAR
```

Alle numerischen Variablen erhalten den Wert 0, alle String-Variablen werden zu Leer-Strings. Felder werden gelöscht, und ihre Dimensionierung wird aufgehoben. CLEAR darf nicht in Prozeduren oder FOR/NEXT-Schleifen verwendet werden. Bei Programmstart wird CLEAR automatisch ausgeführt.

CLR — Einzelvariablen löschen

```
CLR Var [,Var%,Var$,...]
```

Es kann eine Liste von Variablen (keine Feldvariablen) übergeben werden, deren Inhalte gelöscht werden sollen, z.B.:

```
CLR A$,B%,C,D!
```

entspricht:

```
A$=""   B%=0   C=0   D!=0
```

CLS — Bildschirm löschen

```
CLS [#Kanal]
```

Löscht den Ausgabebildschirm bzw. das jeweils geöffnete Intuition-Fenster und setzt den Cursor auf Home (linke obere Ecke). Wenn durch die Option Kanal CLS in eine Diskettendatei geschrieben wird, wird beim Lesen dieser Datei der Bildschirm gelöscht, sobald der Lesezeiger auf CLS trifft. Die Ausgabe eines CHR$(12) mittels PRINT bewirkt das gleiche und wird auch bei der Datei-Ausgabe verwendet.

NEW — Programmspeicher löschen

```
NEW
```

Löscht den BASIC-Arbeitsspeicher mitsamt dem Programm und seinen Variablen. Der Speicher ist für neue Anwendungen frei.

12.3 Zeitoperationen

DATE$ — Systemdatum ermitteln

```
Var$=DATE$
DATE$="Datum-String" (nur V3.0)
```

DATE$ ist eine reservierte String-Variable, die dieses aktuelle Systemdatum als Text-String im Format DD.MM.YYYY (D = Tag/ M = Monat/ Y = Jahr) enthält. Es ist möglich, DATE$ ein neues Datum in "Datum-String" zuzuweisen. Das Format dieses Strings ist unter SETTIME beschrieben.

Wie Sie sicher wissen, kann im Programm Preferences der Workbench das aktuelle Datum eingegeben werden. Diese Angabe wird systemintern ständig auf den aktuellen Stand gebracht. Haben Sie keine Veränderungen an dieser Einstellungen vorgenommen (siehe SETTIME), so erhalten Sie immer das jeweilig auf der Workbench-Diskette abgespeicherte Datum, das nur bei einer akkugepufferten Uhr in der Startup-Sequence neu gesetzt werden kann.

DELAY { DELA } — 1/1-Sek.-Wartefunktion

```
DELAY Sekunden
```

Version 3.0

Sekunden bestimmt, wie viele Sekunden das Programm pausieren soll (sonst siehe PAUSE).

PAUSE { PA } — 1/50-Sek.-Wartefunktion

PAUSE Dauer

Dauer bestimmt in 50stel Sekunden, wie lange das Programm pausieren soll. In dieser Zeit ist ausschließlich die Break-Funktion aktiv.

Andere Aktivitäten (ON ERROR GOSUB, ON MENU xxxx GOSUB, EVERY/AFTER GOSUB etc.) werden für die angegebene Dauer eingestellt.

Das Problem der 16-Bit-Computer (der 32-Bit-Computer erst recht) ist nicht mehr, daß sie so langsam sind und dadurch zwangsläufig Pausen schaffen, sondern daß sie so schnell sind, daß man ihnen manchmal eine kleine Verschnaufpause aufzwingen muß, um bestimmte Programmläufe überhaupt noch kontrollieren zu können. In einigen Programmen finden Sie den Begriff "kleine Klickpause". Er kennzeichnet, was gemeint ist.

Stellen Sie sich vor, Sie lassen eine Schleife mit der Abbruchbedingung EXIT IFMOUSEK=1 enden, und ein daran anschließender Block wird nur betreten, IF MOUSEK=1 ist. Selbst wenn dazwischen noch weitere Zeilen liegen, ist der Computer so schnell, daß der Mausklick zum Verlassen der Schleife gleichzeitig als Bedingungs-Erfüllung für den Eintritt in den IF MOUSEK=1-Block gewertet wird. Damit der Benutzer des Programms in solchen Fällen Zeit hat, die Maustaste wieder loszulassen, legt man zwischen die beiden Bedingungen eine kleine Pause (z.B. PAUSE 5), die dann im Pogrammverlauf von einem "Unwissenden" gar nicht bemerkt wird.

SETTIME { SETT } Uhrzeit und Datum einstellen

```
SETTIME Zeit$,Datum$
```

In Zeit$ und Datum$ werden die neue Systemzeit und das neue Systemdatum bestimmt. Es müssen beide Strings übergeben werden.

Europa-Format :

```
Zeit$ =     "hh:mm:ss"  oder "hhmmss"
Datum$ =    "dd.mm.yyyy"  oder  "dd.mm.yy"
```

USA-Format (nur in V3.0 - siehe MODE):

```
Zeit$ =     "hh:mm:ss"  oder "hhmmss"  (wie oben)
Datum$ =    "mm/dd/yyyy"  oder  "mm/dd/yy"
```

Die Jahresangabe kann auch zweistellig erfolgen (z.B. 86 für 1986), falls es sich um eine Angabe zwischen 1980 und 2079 handelt. Bei der Zeitangabe können die Sekunden (ggf. inkl. Doppelpunkt) weggelassen werden. Die Sekunden werden dann auf Null gesetzt, z.B.:

```
SETTIME "15:37",                  => verändert nur die Uhrzeit
SETTIME "15:37:22","15.07.88"     => verändert beide Einträge
SETTIME "","15.07.1988"           => verändert nur das Datum
```

Wird das Format nicht korrekt eingehalten, werden die Angaben ignoriert, und der alte Inhalt wird unverändert beibehalten. Die Sekunden der Zeitangabe werden übrigens nur in Zweierschritten (0, 2, 4, 6 etc.) übernommen, ungerade Angaben werden auf die nächsthöhere gerade Zahl "gerundet".

TIMES$ System-Uhrzeit ermitteln

```
Var$=TIME$
TIME$="Zeit-String"
```

TIME$ ist eine reservierte String-Variable, die die aktuelle Uhrzeit als Text-String im Format hh:mm:ss (h = Stunde/m = Minute/s = Sekunde) enthält. In V3.0 ist es möglich, TIME$ eine neue Uhrzeit in Zeit-String zuzuweisen. Das Format dieses Strings ist unter SETTIME beschrieben. Die Sekunden der Zeitangabe werden übrigens in Zweierschritten erhöht.

Im GFA-Editor der V3.0 ist dies die Zeitangabe, die Sie rechts oben auf dem Bildschirm sehen. Für Dauer-Computerer (wie mich) wäre es übrigens sicher angebracht gewesen, zusätzlich noch eine Weck-Zeit-Eingabe zu installieren, damit man nicht das Schlafengehen "verschläft". Bei mir müßte dann allerdings nach Erreichen der Weckzeit ein Guru eingebaut werden, da ich bis jetzt jedes gesetzte Zeitlimit um Längen geschlagen habe.

TIMER Laufzeit ermitteln

```
Var=TIMER
```

Reservierte Variable. Enthält die seit Systemstart verstrichene Zeit in 200stel Sekunden.

Der Amiga verfügt über einen Zeit-Zähler, der alle 1/200stel Sekunden um 1 erhöht wird. Dieser Zähler beginnt zum Zeitpunkt des Systemstarts bei Null und erhöht sich also in jeder Sekunde um den Wert 200. Dabei ist es unerheblich, ob zwischenzeitlich irgendwelche Anwendungen ausgeführt werden. Der Zähler orientiert sich an dem konstant bleibenden Takt des Prozessors. D.h. also, daß Sie anhand dieses Zählers exakt feststellen können, wieviel Zeit seit dem Systemstart vergangen ist. Außerdem läßt sich dieser TIMER hervorragend dazu ver-

wenden, von einer bestimmten Zeitdauer abhängige Arbeiten ausführen zu lassen oder die Zeitdauer bestimmter Prozesse zu messen (Benchmark-Tests).

Beispiel 1 (Zeitanzeige):

```
Time=Timer                ! Timer puffern
Do                        ! Endlos-Schleife
  X%=(Timer-Time)/200     ! Differenz in Sekunden
  Print At(10,10);Right$(String$(3,"0")+Str$(X%),3);" Sek."
Loop
Beispiel 2 (Zeitmessung):
Time=Timer
Print "20000er Integer-FOR..NEXT-Leerschleife: ";
For I%=0 To 20000
Next I%
Print (Timer-Time)/200;" Sek."
Time=Timer
Print "20000er Real-FOR..NEXT-Leerschleife: ";
For I=0 To 20000
Next I
Print (Timer-Time)/200;" Sek."
Time=Timer
Print "20000er Integer-REPEAT..UNTIL-Leerschleife: ";
Clr I%
Repeat
  Inc I%
Until I%=20000
Print (Timer-Time)/200;" Sek."
Time=Timer
Print "20000er Integer-DO..LOOP-Leerschleife: ";
Clr I%
Do
  Inc I%
  Exit if I%=20000
Loop
Print (Timer-Time)/200;" Sek."
Beispiel 3 (Quasi-Multitasking):
Print "Bitte <Tasten> drücken"
Graphmode 3                    ! XOR-Modus für Auf/Zu-Box
Do
  Key$=Inkey$                  ! Tatstatur abfragen
  If Key$>""                   ! Taste gedrückt?
    Print Key$;                ! Zeichen ausgeben
  Endif
  If Timer Mod 100=0           ! Jede 1/2 Sekunde
    For I%=10 To 100 Step 6    ! ---------.
      Box 110-I%,110-I%,120+I%,120+I%     |   Parallelprozeß-
    Next I%                               |-   Prozeß
    For I%=100 To 10 Step -6              |    laufen lassen
```

```
      Box 110-I%,110-I%,120+I%,120+I%
    Next I%                         !
  Endif
Loop
```

12.4 Fehlerbehandlung

ERR — Fehler-Code ermitteln

```
Var=ERR
```

ERR ist eine reservierte Variable, die nach Auftreten eines Fehlers seine Identifikationsnummer enthält. Siehe Fehlerliste im Anhang.

ERR$ — Fehlertext liefern

```
Var$=ERR$(Index)
```

Version 3.0

Die Funktion ERR$ liefert in V3.0 den Text der Fehlermeldung, deren Fehlerindex angegeben wurde.

ERROR { ER } — Fehler simulieren

```
ERROR Fehlernummer
```

Fehlernummer steht für die Identifikationsnummer des zu simulierenden Fehlers.

Es wird entweder die entsprechende Fehlermeldung ausgegeben und das Programm beendet, oder es wird - wenn ON ERROR GOSUB aktiv ist - zu der dort angegebenen Prozedur verzweigt.

FATAL — Fehlerart ermitteln

```
Var=FATAL
```

Reservierte Variable. Es wird eine Unterscheidung zwischen "Normal"- und "System-Fehler"-Fehlern getroffen. Ist ein "System-Fehler" aufgetreten (Adresse des zuletzt ausgeführten BASIC-Befehls ist nicht mehr bekannt), enthält sie eine -1. Bei allen anderen Fehlern enthält sie eine 0.

0 -- Allgemeiner BASIC-Fehler. Die Variablen- und Label-Adressen sowie der GOSUB-Stapel sind intakt.

-1 -- Fataler Systemfehler. Die Adressenlage ist zerstört, was normalerweise eine Guru-Meditation zur Folge hat. In GFA-BASIC wird versucht, diese Fehler ebenfalls abzufangen, da das System bei den "normalen" System-Errors die Registerinhalte in einen besonderen Bereich rettet, der - falls kein Total-Absturz eingetreten ist - zur Re-Initialisierung wieder ausgelesen werden kann.

ON ERROR [GOSUB] — Verzweigung bei Fehler

```
ON ERROR GOSUB Prozedur
ON ERROR
```

Verzweigt im ersten Fall zur der angegebenen Prozedur, sobald ein System- oder BASIC-Fehler auftritt. In diesem Fall wird keine Fehlermeldung ausgegeben, sondern das Error-Handling wird dem Programmierer überlassen. So ist es möglich, anhand der Fehlernummer ERR (und ERR$) eigene Fehlermeldungen auszugeben oder das Programm - Ihren Vorstellungen und dem aktuellen Error entsprechend - weiterverzweigen oder einfach fortfahren zu lassen (siehe RESUME).

Wurde zu einer Fehler-Routine verzweigt, schaltet der Interpreter die Fehlerbehandlung nach Abarbeiten der Prozedur wieder in den Normalmodus zurück. Soll also wieder ON ERROR GOSUB aktiviert werden, so muß in der Abfangroutine vor dem RESUME-Sprung erneut eine/die Fehlerroutine angegeben werden.

Die zweite Syntaxvariante schaltet den normalen Error-Modus wieder ein. Es erscheint bei Errors also wieder die übliche Fehlermeldung (in Kompilaten entsprechende OPTION einsetzen!) und das Programm wird daran anschließend abgebrochen.

In der vorliegenden Version 3.0 kann der Befehlsteil GOSUB weggelassen werden. Er wird vom Editor selbständig eingefügt. Beispiel unter FATAL.

RESUME { RESU } Programm nach Error-Routine fortsetzen

```
RESUME        => Nach Error-Routine Programmfortsetzung mit Wiederholung
                 der fehlerhaften Zeile.
RESUME NEXT   => Nach Error-Routine Programmfortsetzung mit der Zeile,
                 die der fehlerhaften Zeile folgt.
RESUME Label  => Nach Error-Routine Programmfortsetzung mit der
                 angegebenen Label-Zeile.
```

Bestimmmt, mit welcher Programmzeile das Programm nach Auftreten eines selbst verwalteten Fehlers (siehe ON ERROR GOSUB) fortgesetzt werden soll. RESUME ist nur innerhalb von Prozeduren zulässig. Sinnvollerweise wäre das eine Fehlerbehandlungsroutine, aber es ist auch möglich, RESUME Label als GOTO-Sprung aus einer Prozedur heraus zu mißbrauchen. GOTO selbst ist ja aus Prozeduren heraus nicht erlaubt, z.B.:

```
Label:              ! Sprung-Label
Print 111           ! PRINT irgendwas
@Routine            ! Routine aufrufen
Procedure Routine
  Resume Label      ! Sprung zum Label
Return
```

Befindet sich bei der Label-Variante das angegebene Label außerhalb der Routine, in der das RESUME- Label steht, wird grundsätzlich der GOSUB-Sprungstapel gelöscht, und alle globalen Variablen werden restauriert.

Nach FATAL-Errors (siehe FATAL) ist ausschließlich RESUME Label zu verwenden. RESUME NEXT und RESUME könnten in diesem Fall evtl. zum Absturz führen.

Ein Beispiel finden Sie unter FATAL.

12.5 Auskünfte

CRSCOL — Aktuelle Cursor-Spalte liefern

```
Var=CRSCOL
```

Reservierte Variable, die die Cursor-Spalte des Windows enthält, in der sich der Cursor aktuell befindet (CRSCOL = CuRSorCOLumn).

Im Gegensatz zur Funktion POS(), die die zeilenbezogene Cursor-Position liefert, kann mit CRSCOL die Spaltenposition des Cursors auf dem Bildschirm (Hires/Lowres: 1 - 80/ 1 - 60) ermittelt werden. In Verbindung mit CRSLIN bildet CRSCOL das Gegenstück zu PRINT AT(S,Z). Ein Beispiel hierzu finden Sie unter CRSLIN.

CRSLIN — Aktuelle Cursor-Zeile liefern

```
Var=CRSLIN
```

Reservierte Variable, die die TOS-Cursor-Zeile enthält, in der sich der Cursor aktuell befindet (CRSLIN = CuRSorLINe).

CRSLIN liegt beim Standard-Font in allen Auflösungen immer im Bereich von 1 - 25. Wenn Sie in Hires die Prozedur Sysfont (siehe ASC()) mit Font = 1 einsetzen, liegt CRSLIN im Bereich von 1 - 50.

In Verbindung mit CRSCOL bildet CRSLIN das Gegenstück zu PRINT AT(S,Z).

Beispiel 1:

```
FOR i=1 TO 500              ! 500mal
  PRINT "CRSCOL/CRSLIN-Test  "; ! Text ausgeben
  IF CRSCOL>50              ! Cursor hinter 50. Spalte?
    PRINT                   ! Dann neue Zeile
  ENDIF
  IF CRSLIN=25              ! Cursor in unterster Zeile
    CLS                     ! Dann Bildschirm löschen
  ENDIF
NEXT i
```

Beispiel 2:

```
a$="TEXT RÜCKWÄRTS"
PRINT AT(37,1);                   ! Cursor positionieren
DO                                ! Zeilen-Schleife
  PRINT AT(37,CRSLIN+1);          ! Cursor neu positionieren
  DO                              ! Spalten-Schleife
    PRINT AT(CRSCOL-2,CRSLIN);    ! Cursor 2 Spalten nach links
    PRINT MID$(a$,(36-CRSCOL),1); ! Zeichen ausgeben
    EXIT IF CRSCOL<22             ! Cursor vor der 22. Spalte =
Exit
  LOOP
  EXIT IF CRSLIN>23               ! Cursor unter 23. Zeile = Exit
LOOP
```

FRE() — Freien Speicherplatz ermitteln

```
Var=FRE(Dummy)
Var=FRE()
```

Es wird eine Garbage-Collection (Müll-Sammlung) durchgeführt und anschließend die Größe des noch freien Speichers geliefert. Dummy ist ein Integerwert ohne Bedeutung.

Der Interpreter sorgt sich ständig um den verbleibenden Speicherplatz. Dazu werden intern die nicht mehr benötigten Variablenbereiche (z.B. gelöschte String-Variablen und Felder) ermittelt, entfernt und die benachbarten Speicherbereiche zusammengeschlossen. Diese Arbeit wird vom BASIC sporadisch erledigt. Zu erkennen ist sie manchmal an kurzen "Aussetzern", bei denen man den Eindruck bekommt, daß das Programm für eine 10tel Sekunde "stottert".

Um diese Garbage-Collection gezielt einsetzen zu können, kann FRE() verwendet werden, auch wenn man am verbleibenden Speicherplatz gar nicht interessiert ist. Ratsam ist der Einsatz von FRE() vor allem direkt vor einer Abfrage von Variablenadressen durch VARPTR() oder ARRPTR(), da die Adreßlage der Variablen (vor allem der Strings) durch eine zwischenzeitlich intern ausgeführte Garbage-Collection verschoben worden sein kann.

In der Amiga-Version 3.0 ist Dummy optional. Wird eine Leerklammer angegeben, wird vor der Speicherplatz-Ermittlung keine Garbage-Collection durchgeführt.

TYPE() — Variablentyp ermitteln

```
Var=TYPE(Pointer)
```

Pointer steht für einen *-Pointer (z.B. PRINT TYPE(*Var%)). Es wird der Typ der dadurch repräsentierten Variablen geliefert.

Type:

- -1 = Fehler aufgetreten
- 0 = Realzahlvariable (Var)
- 1 = String-Variable (Var$)
- 2 = 4-Byte-Integervariable (Var%)
- 3 = Boolesche Variable (Var!)
- 4 = Real-Feldvariable (Var())
- 5 = String-Feldvariable (Var$())

6 = 4-Byte-Integer-Feldvariable (Var%())
7 = Boolesche Feldvariable (Var!())

Ab hier nur für V3.0:

8 = 2-Byte-Integervariable (Var&)
9 = 1-Byte-Integervariable (Var|)
12 = 2-Byte-Integer-Feldvariable (Var&())
13 = 1-Byte-Integer-Feldvariable (Var|())

Beispiel:

```
A=10.1
A$="ABC"
A%=10
A!=-1
Print Type(*A)'Type(*A$)'Type(*A%)'Type(*A!)
```

Ein weiteres Beispiel finden Sie unter * (Pointer).

12.6 Multitasking

AFTER x GOSUB { AF }	Single-Interrupt-Routinenaufruf

`AFTER Ticks [GOSUB] Prozedur` *Version 3.0*

Ruft nach Ticks Zeiteinheiten (1 Sekunde = 200 Ticks) die angegebene Prozedur einmal auf. Der Befehl wird jedoch - generell - immer erst nach vollständiger Abarbeitung eines BASIC-Befehls ausgeführt. Wenn z.B. durch SOUND x,x,x,x,Dauer, PAUSE, DELAY, INP() INPUT, BLOAD etc. die Programmausführung unterbrochen wird, so wird der Sprung zur angegebenen Prozedur ggf. erst bei Wiederaufnahme des Programms wirksam. Beispiel:

```
AFTER 100 GOSUB abc  ! AFTER-Aufruf
t%=TIMER
INPUT a$             ! AFTER-Ausführung wird unterbrochen
DO
LOOP
```

```
PROCEDURE abc
  PRINT "AFTER 100 GOSUB erst nach ";TIMER-t%;
  PRINT " Ticks ausgeführt"
  END
RETURN
```

Anmerkung: Es ist bisher nicht möglich, mehrere AFTER-Interrupt-Anweisungen gleichzeitig zu verarbeiten. Die zuletzt verfügte AFTER GOSUB- bzw. EVERY GOSUB-Anweisung hebt also die vorangegangene auf.

Beispiel:

```
AFTER 1000 GOSUB aa  ! Erster Aufruf
AFTER 400 GOSUB bb   ! Zweiter Aufruf
DO                   ! Endlos-Schleife
LOOP
PROCEDURE aa         ! Prozedur für ersten Aufruf
  PRINT 11111
RETURN
PROCEDURE bb         ! Prozedur für zweiten Aufruf
  PRINT 22222
RETURN
```

Hier im Beispiel wird nur einmal nach zwei Sekunden die Prozedur des zweiten Aufrufs angesprungen. Der erste Aufruf ist durch den zweiten ungültig geworden. PRINT 11111 wird also nicht mehr ausgeführt.

Es gibt allerdings die Möglichkeit, aus einer AFTER x GOSUB-Anweisung eine zweite EVERY x GOSUB-Anweisung zu machen. Dazu muß ganz einfach die entsprechende AFTER x GOSUB-Prozedur vor ihrem RETURN wieder neu initialisiert werden. Dabei ist zu beachten, daß so nicht der gleiche regelmäßige Rhythmus wie bei EVERY zu erreichen ist, da ja die Initialisierung selbst immer wieder Zeit in Anspruch nimmt. Eine solche - zu EVERY zusätzliche - Interrupt-Schleife ist jedoch in den meisten Fällen besser als keine.

Beispiel:

```
AFTER 60 GOSUB proc1          ! AFTER initialisieren
EVERY 100 GOSUB proc2         ! EVERY initialisieren
DO                            ! Endlos-Schleife
LOOP
PROCEDURE proc1               ! AFTER-Prozedur
  PRINT "AFTER-Procedure"
  AFTER 60 GOSUB proc1        ! AFTER neu initialisieren
RETURN
PROCEDURE proc2               ! EVERY-Prozedur
  PRINT "EVERY-Procedure"
RETURN
```

AFTER CONT { AF CONT } Single-Interrupt-Routine freigeb.

AFTER CONT *Version 3.0*

Setzt nach AFTER STOP die Single-Interrupt-Kontrolle fort. Weiteres siehe AFTER STOP.

AFTER STOP { AF STOP } Single-Interruptroutine sperren

AFTER STOP *Version 3.0*

Unterbricht die Single-Interruptkontrolle. Durch AFTER STOP kann veranlaßt werden, daß die Ausführung der aktuellen AFTER x GOSUB-Prozedur solange verzögert wird, bis wieder AFTER CONT verfügt wird. Wird AFTER CONT nicht eingesetzt, so bleibt AFTER x GOSUB bis zum Programmende inaktiv - es sei denn, durch einen neuen AFTER x GOSUB wurde ein neuer Timer gesetzt.

Z.B. soll nach 3 Sekunden eine AFTER-Procedure angesprungen werden. Die Ausführung wird jedoch nach 2 Sekunden durch AFTER STOP unterbrochen und nach weiteren 2 Sekunden durch AFTER CONT wieder zugelassen, so wird AFTER x GOSUB erst nach diesen 4 Sekunden ausgeführt, obwohl die

Zeitspanne von 3 Sekunden schon verstrichen ist. Weitere AFTER CONT-Aufrufe für denselben Prozeß haben dann keine Wirkung mehr, da die Routine nur einmal ausgeführt wird.

Beispiel:

```
t%=TIMER
AFTER 600 GOSUB proc1     ! Nach 3 Sekunden
PRINT "AFTER 600 GOSUB ist aktiv"
EVERY 400 GOSUB proc2     ! Alle 2 Sekunden
DO                        ! Endlos-Schleife
LOOP
PROCEDURE proc1
  PRINT "AFTER 600 GOSUB nach ";(TIMER-t%);" Ticks ausgeführt!"
  EDIT
RETURN
PROCEDURE proc2
  a%=a% XOR 1             ! A% als Wechsel-Flag (1/0)
  IF a%=1
    PRINT "AFTER 600 GOSUB unterbrochen"
    AFTER STOP            ! AFTER GOSUB unterbrechen
  ELSE
    PRINT "AFTER 600 GOSUB wieder aufgenommen"
    AFTER  CONT           ! AFTER GOSUB fortsetzen
  ENDIF
RETURN
```

EVERY x GOSUB { EV } — Interrupt-Routinenaufruf

```
EVERY Ticks [GOSUB] Prozedur
```

Ruft ständig nach Ablauf von Ticks Zeiteinheiten (1 Sekunde = 200 Ticks) die angegebene Prozedur auf. Der Befehl wird jedoch - generell - immer erst nach vollständiger Abarbeitung eines BASIC-Befehls ausgeführt. Wenn z.B. durch SOUND x,x,x,x,Dauer, PAUSE, DELAY, INP() INPUT, BLOAD etc. die Programmausführung unterbrochen wird, so wird der Sprung zur angegebenen Prozedur ggf. erst bei Wiederaufnahme des Programms wirksam.

GFA-Fans mußten lange auf diesen Befehl warten. Beachten Sie außerdem die Anmerkung zu AFTER GOSUB.

Beispiel:

```
Every 50 Gosub Multi      ! Interrupt initialisieren
Print At(1,3);"Bitte Text eingeben und mit Maus zeichnen."
Print "(Die Zeit-Datum-Anzeige kann durch Klick";
Print "geöffnet und dann geändert werden.)"
Do                        ! Endlos-Schleife
  Keytest A%              ! Tastatur abfragen
  If (A% And 255)         ! ASCII-Taste gedrückt?
    Out 5,A% And 255      ! ASCII über Console ausgeben
  Endif
  If Mousek               ! Maustaste gedrückt?
    Repeat                ! Box-Schleife
      Box Mousex,Mousey,Mousex+10,Mousey+10
    Until Mousek=0        ! Bis Maustaste losgelassen
  Endif
Loop
Procedure Multi           ! EVERY-Prozedur
  Every Stop              ! EVERY unterbrechen (siehe unten)
  Gosub Timdat(1,1,1)     ! Zeit-Prozedur aufrufen
  Every  Cont             ! EVERY fortsetzen   (siehe unten)
Return
```

Die in diesem Beispiel verwendete Zeit-Routine Timdat finden Sie unter SETTIME beschrieben. Sie ist in dieses Demo-Programm vor Start einzubinden.

Hier können Sie einen Effekt beobachten, der mich fast zur Verzweiflung getrieben hat. Ich wußte genau, daß die Timdat-Routine einwandfrei funktionierte. Als ich den Timdat-Aufruf in die Multi-Prozedur blauäugig eingebunden hatte, war ein Probelauf fällig ---!!!--- und es funktionierte gar nichts mehr oder immer, wenn ich das Datum oder die Zeit ändern wollte, brach das Chaos aus.

Ich durchsuchte Timdat nach möglichen Fehlern und suchte und suchte... Als ich dann genug gesucht hatte, überprüfte ich mit TRON den Programmlauf. Da die Interrupt-Prozeduren von TRON ausgeschlossen werden (!), konnte ich so auch nichts erkennen. Nach einiger - haareraufend verbrachter - Zeit kam mir der Geistesblitz, daß ja die Timdat-Routine auch dann immer wieder durch EVERY aufgerufen wird, wenn man innerhalb von Timdat gerade mit der Zeit- oder Datumseingabe beschäftigt ist. Daraus entstand - quasi rekursiv - das Durcheinander.

Erst als ich vor dem Timdat-Aufruf EVERY STOP und danach EVERY CONT einsetzte, klappte alles wieder so, wie es sollte.

EVERY CONT { EV CONT } **Interrupt-Routine freigeben**

```
EVERY CONT
```

Version 3.0

Nimmt nach EVERY STOP die EVERY-Interrupt-Kontrolle wieder auf. EVERY GOSUB wird wieder ausgeführt.

EVERY STOP { EV STOP } **Interrupt-Routine sperren**

```
EVERY STOP
```

Version 3.0

Unterbricht die EVERY-Interrupt-Kontrolle. EVERY GOSUB wird nicht mehr ausgeführt. Fortsetzung erst wieder durch EVERY CONT.

12.7 Debugging

DUMP { DU } **Variableninhalte/Namen ausgeben**

```
DUMP [Defstring$ ] [TO Datei$]
```

Version 3.0

Wird DUMP ohne Optionen verwendet, werden die Inhalte aller Variablen sowie die Dimensionierungen aller Felder ausgegeben.

In Deftsring$ kann optional eine nähere Spezifikation angegeben werden:

"a" Die Inhalte aller Variablen sowie die Dimensionierungen aller Felder, deren Name mit a beginnt, werden ausgegeben.

":" Alle Label-Namen sowie ihre Zeilennummer werden ausgegeben.

":a" Alle Label-Namen, die mit a beginnen, sowie ihre Zeilennummer werden ausgegeben.

"@" Alle Prozedur- und Funktionsnamen sowie ihre Zeilennummer werden ausgegeben.

"@a" Alle Prozedur- und Funktionsnamen, die mit a beginnen, sowie ihre Zeilennummer werden ausgegeben.

Ausgegebenen Prozedurnamen wird als Kennung @ angehängt, sowie dahinter - falls die Prozedur im aktuellen Programm noch existiert - die Nummer der Zeile, in der die Prozedur beginnt. Dasselbe geschieht bei Namen noch existierender Funktionen. Diese erhalten jedoch als Kennung ein FN. Die Namen von String-Funktionen werden zusätzlich noch mit einem $ gekennzeichnet.

Prozeduren, Funktionen und Label, die definiert waren und wieder aus dem Programm gelöscht wurden, werden ohne Zeilennummern dargestellt. Dagegen sind nicht mehr existierende Variablennamen - die ebenfalls ausgegeben werden, solange sie noch in der internen Liste stehen - auf den ersten Blick nicht zu erkennen. Das einzige Merkmal ist, daß sie keinen Inhalt haben (0 bzw. ""), was jedoch bei noch existierenden Variablen ebenso der Fall sein kein.

Durch Save,A, New und anschließendes Merge werden die überflüssigen Namen aus der internen Referenzliste entfernt.

Bei String-Variablen werden max. 60 Zeichen ihres Inhalts ausgegeben, die dann in " (Anführungszeichen) eingeschlossen sind.

Ist der Inhalt länger als 60 Zeichen, steht am Textende das Zeichen >. Enthält der String Zeichen mit einem ASCII-Wert unter 32, werden diese Zeichen durch einen Punkt . repräsentiert. Es kann außerdem - unabhängig von der Option Deftstring$ - hinter dem Zusatz TO in Datei$ der Name einer Datei angegeben werden, in die dann die DUMP-Ausgaben umgeleitet werden. Diese Datei erhält - sofern keine andere angegeben wird - die Extension .DMP., ggf. schon existierende .DMP-Dateien gleichen Namens erhalten die Backup-Extension .BAK.

TRACE$ — Im Trace-Modus aktuelle Befehlszeile liefern

```
Var$=TRACE$
```

Version 3.0

Innerhalb der durch TRON Proc bestimmten Prozedur kann durch die reservierte Variable TRACE$ der Text der aktuellen Programmzeile vor ihrer Ausführung ermittelt werden. Außerhalb dieser Prozedur ist TRACE$ ohne Inhalt.

Ein Beispiel zu TRACE$ finden Sie unter TRON Proc.

TROFF { TROF } — Trace-Modus ausschalten

```
TROFF
```

Im Anschluß an TROFF ist das Programm wieder im normalen Ausführungsmodus. TRON - sowie TRON Proc in Version V3.0 - werden ausgeschaltet. TROFF innerhalb einer TRON-Proc-Routine hat keine Wirkung.

TRON Proc { TR } — **Trace-Modus in Prozedur lenken**

`TRON Prozedurname` *Version 3.0*

Dies ist eine V3.0-Variante des bekannten TRON-Befehls. Sie lenkt den Trace-Modus in die durch Prozedurname angegebene Prozedur um. Dabei wird nicht - wie bei TRON sonst üblich - automatisch die aktuelle Befehlszeile ausgegeben, sondern es kann beliebig auf die durch TRACE$ ermittelte, als nächstes auszuführende Befehlszeile oder ggf. auf die durch DUMP ermittelten Variableninhalte reagiert werden.

Auch TRON Proc wird ggf. durch TROFF deaktiviert.

Möchten Sie eine Debugging-Routine selbst schreiben, so kann Ihnen TRON Proc dabei nicht helfen, denn sowohl die Interrupt-Routinen AFTER x GOSUB und EVERY x GOSUB als auch die hier angegebene Debugging-Prozedur werden von der TRACE$-Ausgabe ausgeschlossen.

12.8 Diverses

$ — **Textbereich für V3.0-Compiler deklarieren**

`$ Text` *Version 3.0*

Deklariert beliebige Programmzeilen als Textspeicher zur Weiterverarbeitung durch den V3.0-Compiler. Im Interpreter-Betrieb wird eine solche Zeile wie REM bzw. ' behandelt. Nähere Informationen sind erst mit Erscheinen des V3.0-Compilers zu erwarten.

DEFLIST { DEFLIS } Listing-Format festlegen

```
DEFLIST Format
```

Format:

0 Befehlsnamen werden groß, Variablennamen klein geschrieben (PRINT var$). In V3.0 wird bei global deklarierten Variablennamen das Postfix vernachlässigt (Postfix siehe DEFxxx-Befehle DEFBYT etc.).

1 Anfangsbuchstaben von Befehls- und Variablennamen werden groß geschrieben (Print Var$). In V3.0 wird bei global deklarierten Variablennamen das Postfix vernachlässigt.

DEFLIST kann nur im Interpreter-Direktmodus verwendet werden.

DEFNUM { DEFN } Rundung von Ziffern-Ausgaben

```
DEFNUM Stelle
```

Stelle gibt die Ziffernstelle an (3 - 13), auf die alle folgenden - durch PRINT etc. - auszugebenden Werte gerundet werden sollen. Diese Einstellung wird erst durch den nächsten DEFNUM-Befehl verändert. Durch Angabe der maximalen Stellenanzahl (13) wird die normale Werteausgabe wieder angeschaltet.

Bei Realzahlen wird der Vorkomma-Anteil, der ggf. hinter Stelle liegt, als Nullen ausgegeben. Liegt Stelle im Nachkommabereich, werden alle dahinterliegenden Nachkommastellen ignoriert, bzw. die nächste Stelle hinter Stelle wird zur Rundung verwendet.

Die Rundung erfolgt mathematisch exakt (Int(A+0.5)). In diesem Format ausgegebene Variableninhalte bleiben von DEFLIST unberührt. Die interne Rechengenaugkeit wird hierdurch nicht beeinträchtigt. Beispiel:

```
a=RANDOM(1000000)+RND
PRINT "Originalwert: ";a;CHR$(10)
FOR i%=3 TO 11
  DEFNUM i%
  PRINT a,"DEFNUM ";i%
NEXT i%
```

FALSE — Unwahr-Konstante

```
Var=FALSE
```

Reservierte Variable. Enthält konstant den Un-Wert 0.

LET { LE } — Daten zuweisen

```
LET Var=Wert
LET Var$=Text
```

Es wird Wert bzw. Text der angegebenen Variablen zugewiesen. Der LET-Befehl ist eigentlich überflüssig, da es bei Variablennamen keine Einschränkungen (außer PI, ERR, FATAL, TIMER etc.) mehr gibt und so die Variablen von der Syntax her deutlich sind. Hier hat LET nur noch den Zweck der Kompatibilität zu den BASIC-Dialekten (z.B. AmigaBASIC), deren Programme man durch Merge in den GFA-Arbeitsspeicher laden kann.

MODE { MOD } Zahlen/Datum -> Europa/USA wählen

```
MODE Modus
```

Version 3.0

In der Grundeinstellung werden in GFA-BASIC durch PRINT USING Werte im Europa-Format ausgegeben. D.h. ggf., daß als Tausendertrennung ein Komma und als Dezimaltrennung ein Punkt verwendet wird. In den USA gelten die umgekehrten Konventionen. Ähnliches gilt für die Darstellung des Datums (siehe SETTIME).

Modus ist als 2-Bit-Vektor anzusehen, dessen Bit 0 die Darstellung des Datums durch DATE$ und FILES bzw. über das Eingabeformat des Datums bei SETTIME und DATE$= bestimmt. Bit 1 entscheidet über die zu verwendende Art der Werte-Darstellungsart bei PRINT USING-Ausgaben bzw. bei Verwendung von STR$().

Modus	USING	DATE$
&X00	#,###.##	25.05.1988
&X01	#,###.##	05/25/1988
&X10	#.###,##	25.05.1988
&X11	#.###,##	05/25/1988

TRUE Wahr-Konstante

```
Var=TRUE
```

TRUE ist eine reservierte Variable, die konstant den Wert -1 enthält. Bei verschiedenen Funktionen (z.B. EXIST) wird ein Wahrheitswert zurückgegeben. Zur besseren Übersichtlichkeit kann bei Bedingungsabfragen dieser Art, bei Bit-Flag-Verwaltung oder Zuweisungen etc. statt des Wertes -1 die Konstante TRUE verwendet werden (siehe auch FALSE).

Beispiel 1:

```
FILESELECT "FileBox","Laden","SYS:",F$     ! Datei wählen
IF EXIST(F$)=TRUE                          ! Datei auf Disk?
  ... weiteres
  ... Programml
```

Beispiel 2:

```
DO                                ! Endlos-Schleife
  IF MOUSEK=1                     ! Linke Maustaste gedrückt?
    bitflag!=bitflag! XOR TRUE    ! Flag bei jedem Mausklick
    '                             ! Abwechselnd an- und ausschalten
    onoff%=ABS(bitflag!=TRUE)     ! Ergibt: An = 1/Aus = 0
    PRINT AT(1,1);RIGHT$("an ",3*onoff%)
    PRINT AT(1,1);RIGHT$("aus",3*ABS(onoff%=FALSE))
    PAUSE 6                       ! Kleine Klickpause
  ENDIF
  CIRCLE MOUSEX*onoff%,MOUSEY*onoff%,10*onoff%
LOOP
```

Vielleicht können Sie im obigen Beispiel auf Anhieb nicht viel mit den beiden PRINT-Zeilen anfangen. Dadurch, daß ich eine Wahrheitsabfrage in die Zeilen mit einbaue, erspare ich mir eine entsprechende IF-Abfrage. Die Variable Onoff% enthält, je nachdem, ob das Bitflag! -1 oder 0 ist, den Wert 1 oder 0. Mit der RIGHT$-Konstruktion lasse ich nur dann die Ausgabe des Strings zu, wenn der Wert 3 (Länge des Strings) mit 1 multipliziert wird. Steht in Onoff% der Wert Null, ist die zu ermittelnde RIGHT$-String-Länge ebenfalls Null und es wird kein String ausgegeben.

Im Normalverfahren würde die Konstruktion so aussehen:

```
IF Bitflag!=TRUE
  PRINT AT(1,1);"an "
ELSE
  PRINT AT(1,1);"aus"
ENDIF
```

SWAP { SW } Variablen/Felder/Pointer tauschen

```
SWAP Var1,Var2
SWAP Element(x),Element(y)
SWAP Feld1(),Feld2()
SWAP *Pointer,Feld()
```

Dies ist der SWAP-Befehl, in V3.0 gibt es SWAP() auch als Funktion. SWAP tauscht die Inhalte zweier gleichartiger Variablen, Felder oder Feldelemente (numerische oder alphanumerische) bzw. einen Zeiger auf einen Feld-Descriptor (siehe *) mit dem Descriptor des angegebenen Feldes (Feldübergabe an Prozeduren). Bei Feldern wird auch die Dimensionierung vertauscht. Es ist bei Pointer/Feld-SWAP nicht nötig, das angegebene Feld vorher zu dimensionieren.

Beispiel 1:

```
A%=100
B%=1000
Print "A%, B% vor SWAP:";A%'B%
Swap A%,B%
Print "A%, B% nach SWAP:";A%'B%
```

Beispiel 2:

```
Dim A%(10)
@Routine(*A%())
Procedure Routine(X%)
  Swap *X%,Dummy%()
  .
  ... Ab hier ist das vorherige Feld A%() unter
  ... dem Namen Dummy%() ansprechbar. Feld A%()
  ... ist leer.
  .
  ... Routinentext
  .
  Swap *X%,Dummy%()
  .
  ... Ab hier ist Feld Dummy%() wieder Feld A%()
  ... und Feld Dummy%() ist wieder leer.
  .
Return
```

Diese indirekte Übergabe von Feldern an Prozeduren hat den Vorteil, daß das durch SWAP in die Prozedur "hineingetauschte" Feld nicht unter seinem eigentlichen (globalen) Namen angesprochen werden muß, sondern innerhalb der Prozedur unter einem lokalen Namen angesprochen und bearbeitet werden kann.

Genaugenommen ist diese Konstruktion in der Version 3.0 jedoch überflüssig, da hier Felder genauso wie Variablen durch VAR direkt an die Prozedur übergeben werden können:

```
DIM feld%(1)
xyz(feld%())
PRINT Feld%(1)
PROCEDURE Xyz(VAR Dummyfeld%())
  Dummyfeld%(1)=100
RETURN
```

VOID { V } — Dummy-Zuweisung

```
VOID Funktion
```

VOID ist ein Ersatz für eine sogenannte Dummy-Zuweisung, jedoch - vor allem in Compilaten - erheblich schneller. Es wird eine Funktion aufgerufen, ohne dieser eine Rückgabe-Variable zur Verfügung stellen zu müssen. In vielen Fällen ist die Zuweisung des Funktionsergebnisses an eine Aufnahme-Variable bei Funktionsaufrufen unnötig, da das Ergebnis nicht von Interesse ist. Um Speicherplatz (und Zeit) zu sparen, können diese Funktionen mit VOID aufgerufen werden.

Beispiel:

```
Statt:  A=INP(2)          ! Auf Taste warten
        -->  VOID INP(2)
Statt:  A=FRE(0)          ! Garbage-Collection
        -->  VOID FRE(0)
```

13. Interaktionen (Programm/Benutzer)

ALERT { A } — Alert-Box erstellen

```
ALERT Icon%, Boxtext$, Def_Button%, Buttontext$, Backvar%
```

Diese an Intuition angelehnte Funktion erlaubt es, mit dem Anwender unkompliziert zu kommunizieren. Prinzipiell lassen sich damit sogar umfangreiche Menüs verwalten. Da die Alert-Box (wachsam,alarmbereit) aber nur maximal drei Auswahlmöglichkeiten zur Verfügung stellt, ist sie eher für kleinere Abfragen oder Hinweistexte geeignet.

Icon%

Stellt eine Zahl dar, unter der man ein Symbol abrufen kann, das den Charakter der Nachricht kennzeichnen soll. Dies ist aber leider in der Amiga-Version 3.0 noch nicht implementiert.

Boxtext$

Hier wird der eigentliche Text (Mitteilung/Frage) an die Funktion übergeben. Das Pipe-Zeichen | gilt darin als Trennungszeichen zwischen den einzelnen Zeilen. Es können insgesamt 5 Zeilen zu maximal je 40 Zeichen dargestellt werden. Der Text kann direkt in den Befehl geschrieben, als String-Ausdruck oder auch als String-Variable übergeben werden.

Def_Button%

Es wird die Nummer (1,2,3) des Buttons übergeben, der außer durch Mausklick auch durch die <Return>-Taste (default) bestätigt werden kann. Dieser Button wird in der Box stark umrandet gezeichnet (0 = kein Default-Button).

Buttontext$

Hierdurch erfolgt die Beschriftung der Buttons. Auch hier gilt das |-Zeichen als Trennstrich zwischen den einzelnen Button-Texten.

Backvar%

Numerische Variable, in der der Befehl die Nummer des bestätigten Button (1,2,3) zurückgibt.

Beispiel:

```
boxtext$="Klicken Sie bitte eines|der unteren Kästchen an!"
boxtext$=b_oxtext$+"|Ich sage Ihnen dann, welches|es war!"
buttontext$="XX 1 XX|XX 2 XX|XX 3 XX"
ALERT 0,boxtext$,0,buttontext$,backvar%
boxtext$="Es war :          | |das Kästchen "+STR$(backvar%)
buttontext$=" OKAY | OKAY | OKAY "
ALERT 0,boxtext$,2,buttontext$,backvar%
```

Diese Form der Alert-Box ist aus Gründen der Sicherheit nicht besonders variabel. Vom Interpreter werden sämtliche Eingaben auf ihre Zulässigkeit überprüft. Werden z.B. mehr als 40 Zeichen Boxtext in einer Zeile angegeben, wird die Zeile vom Interpreter nach dem 40sten Zeichen abgeschnitten.

Wer die eigentliche Request-Box kennt, der wird verstehen, daß man Einschränkungen in der Bemessung gemacht hat.

FILESELECT { FILE } — Datei auswählen

```
FILESELECT "Titel","OKText","Pfad",Backvar$
```

Erstellt ein Dialog-Formular zur Dateiauswahl und liefert den gewählten Dateinamen (ggf. inkl. Pfad).

Bei Rückkehr zum Programm sind vier verschiedene Eintragsmöglichkeiten in Backvar$ möglich:

1. Wurde vom Anwender eine Datei gewählt, steht ihr Name auch anschließend zusammen mit dem vollständigen Pfad in Backvar$.

2. Wurde die OK-Box ohne Auswahl bedient, gibt es zwei Varianten:

2a. Es wurde durch Auswahl ein Dateiname übergeben, der noch in der Eintragszeile steht und die Ok-Box bedient, dann steht dieser Name oder Pfad-Eintrag auch in Backvar$.

2b. Es wurde keine Auswahl getroffen und auch kein Name übergeben, bzw. die Eintragszeile wurde vom Anwender gelöscht und die OK-Box wurde bedient, dann wird in Backvar$ ein Leer-String zurückgegeben.

3. Wird Abbruch angeklickt, ist Backvar$ absolut leer.

Dies ist einer der wichtigsten Befehle zur Bewältigung von Diskettenoperationen. Die Menge an Funktionen, die dieser so einfach scheinende Befehl in sich vereinigt, kann man nur ermessen, wenn man schon einmal versucht hat, auf die herkömmliche Weise Dateien zu speichern, zu laden, zu löschen oder zu verändern.

Diese Box listet in einem Fenster die unter dem angegebenen Pfadnamen (Zeile unter den File-Einträgen) verfügbaren Dateien auf. Alle mit einem Stern gekennzeichneten Dateien stellen sogenannte Subdirectories (Unter-Inhaltsverzeichnis) dar, in denen weitere Dateien unter dem für dieses Subdirectory typischen Pfadnamen abgespeichert sein können. Durch Anklicken des Sub-Dir-Namens wird es geöffnet, und die darin befindlichen Dateien können angewählt werden. Verlassen wird ein Unter-Verzeichnis, indem man das kleine Feld links oben unter der Titelzeile anklickt.

Da GFA-BASIC immer die Disk-Station zur aktuellen erklärt, von der das Programm geladen wurde, kann es notwendig werden, im Pfadnamen die Stationsangabe zu ändern. Sie können

dazu einmal aus den zur Verfügung stehenden Geräten auswählen, die von GFA-BASIC in der Zeile unter der File-Tabelle aufgelistet werden, indem Sie mit dem Mauszeiger in dieser Zeile ein Feld anklicken. Ein anderer Weg ergibt sich durch eigenständige Eingabe des Pfadnamens in dem String-Gadget. Klicken Sie dazu auf dieses Gadget. Nach der Änderung brauchen Sie jetzt nur noch die <Return>-Taste zu betätigen, und das neue Directory wird angezeigt.

Es folgt eine fast unscheinbare Routine, die es aber in sich hat. Wie oft stellt sich die Aufgabe, die Richtigkeit einer Dateinamen-Eingabe zu überprüfen oder ein Backup-File anzulegen. Während der eigentliche Dateiname in den allermeisten Fällen für den Anwender frei bestimmbar ist, hat man als Programmierer doch oft ein Interesse daran, daß die Extension richtig gesetzt ist. Um sicherzustellen, daß garantiert die richtige Extension verwendet wird, kann man diese Prozedur folgendermaßen aufrufen:

```
FILESELECT "Titel","OK","RAM:#?.ABC",a$  ! Eingabe des Dateinamens
@extend(a$,"ABC",*n_file$)     ! Extension überprüfen
PRINT n_file$
'
PROCEDURE extend(pr$,ex$,ps%)
  ' Pr$ = Dateipfad u. -name
  ' Ex$ = gewünschte Extension
  ' Ps% = String-Rückgabe
  '
  LOCAL nl%,dn$,i%
  IF RIGHT$(pr$)<>"/" AND pr$>""
    '                          ! Gültiger Dateiname?
    FOR i%=LEN(pr$) DOWNTO 1   ! Namen von hinten aus nach dem
      '                        ! erstem Backslash durchsuchen
      INC nl%                  ! Zeichen mitzählen
      EXIT IF MID$(pr$,i%,1)="/" ! Exit, wenn Backslash erreicht
    NEXT i%
    dn$=RIGHT$(pr$,nl%)        ! Reinen Dateinamen bilden
    IF INSTR(dn$,".")=0        ! Keine Extension enthalten?
      *ps%=pr$+"."+ex$         ! Pfad und Extension zurückgeben
      dn$=dn$+"."+ex$          ! Namen komplettieren
    ENDIF
    IF RIGHT$(dn$,4)<>"."+ex$  ! Falsche Extension?
      IF LEFT$(dn$,1)<>"."     ! Name größer als nur Extension?
        *ps%=LEFT$(pr$,LEN(pr$)-nl%)+LEFT$(dn$,INSTR(dn$,"."))+ex$
        '                      ! Pfad + Name + Extension zurück
      ELSE                     ! Name besteht nur aus Extension!
```

```
        *ps%="000"          ! Unbrauchbaren Namen zurückgeben
      ENDIF
    ELSE                    ! Richtige Extension!
      *ps%=LEFT$(pr$,LEN(pr$)-nl%)+dn$ ! Pfad + Name zurück
    ENDIF
  ELSE                      ! Ungültiger Dateiname!
    *ps%="000"              ! Unbrauchbaren Namen zurückgeben
  ENDIF
RETURN
```

Wurde in diesem Beispiel in der FILESELECT-Box ohne Änderung einfach die OK- oder ABBRUCH-Box angeklickt, erhält man in N_file$ nach Extend-Aufruf den Ausdruck 000. Das gleiche geschieht, wenn vom Anwender entweder die Auswahlzeile ersatzlos gelöscht wurde oder nur die Extension geändert, aber kein Name dazu eingegeben wurde. In allen anderen Fällen wird die Extension des eingegebenen Namens mit der Vorgabe im zweiten Parameter-String des Extend-Aufrufs verglichen und bei Nicht-Übereinstimmung durch die Vorgabe ersetzt. Anschließend erhalten Sie den gesamten Dateinamen mit evtl. geänderter Extension in N_file$ zurück.

Bei Backup-Files kommt noch eine zweite Mini-Routine zum Einsatz. Die Auswirkung einer Backup-Routine begegnet Ihnen immer dann, wenn Sie ein .GFA- oder .LST-File abspeichern, dessen Name bereits auf der Diskette existiert. GFA-BASIC ändert dann automatisch die Extension der bereits bestehenden Datei in .BAK um.

Diese Routine Backup erwartet zwei Parameter-Strings. Der erste gibt den Namen an, unter dem die Datei abgelegt werden soll. Der zweite bestimmt die Extension, die Sie jener Datei geben wollen, die evtl. unter demselben Namen wie die neue Datei schon auf Diskette existiert und nun als Backup gesichert werden soll.

Die Backup-Routine erledigt nun das Finden und Umbenennen mit Hilfe der Extend-Routine, so daß Sie anschließend Ihre neue Datei anlegen können. Als Beispiel lege ich hier die erste BitPlane des Workbench-Screen auf Diskette ab. Der Backup-Effekt wird erst deutlich, wenn Sie das Programm zweimal star-

ten und sich die Dateien mit FILES im Direktmodus anschauen. Sie haben nun zwei Dateien mit gleichem Namen, aber unterschiedlicher Extension.

```
FILESELECT "Backup Test","Test","RAM:",a$ ! Eingabe des Dateinamens
@backup(a$,"BAK")              ! Evtl. Backup anlegen
@extend(a$,"SCR",*n$)          ! Extension überprüfen
IF n$<>"000"                   ! Brauchbarer Pfad + Name?
  BSAVE n$,SCREEN(0)+184,20480! Datei anlegen
ENDIF
'
PROCEDURE backup(pr$,ex$)
  ' Pr$ = Dateipfad u. -name
  ' Ex$ = Backup-Datei-Extension
  LOCAL xn$
  IF EXIST(pr$)               ! Datei auf Diskette vorhanden?
    @extend(pr$,ex$,*xn$)     ! Backup-Extension einbauen
    IF xn$<>"000"             ! Brauchbarer Dateiname?
      IF EXIST(xn$)           ! Schon Backup-File vorhanden?
        KILL xn$              ! Dann löschen
      ENDIF
      NAME pr$ AS xn$    ! Alte Datei auf Backup umbenennen
    ENDIF
  ENDIF
RETURN
```

MOUSE { MOU } — Maus-Status ermitteln (gesamt)

```
MOUSE Xpos,Ypos,Tasten
```

In Xpos und Ypos wird die aktuelle X- bzw. Y-Koordinate des Mauszeigers sowie in Tasten der Status der Maustasten übergeben.

0 = Keine Taste gedrückt				
1 = Linke Taste gedrückt	(Bit 0	->	2^0	= 1)
2 = Rechte Taste gedrückt	(Bit 1	->	2^1	= 2)
3 = Beide Tasten gedrückt	(Bit 0+1	->	2^0+2^1	= 3)
4 = Mittlere Taste gedrückt	(Bit 2	->	^2	= 4)

Als Bit-Vektor betrachtet, sind von Tasten nur die untersten 2 Bits interessant. Dabei steht Bit 0 für die linke Maustaste und Bit 1 für die rechte.

Anmerkung: In Version V3.0 werden (bzw. bei CLIP OFFSET) in Xpos und/oder Ypos auch negative Werte geliefert, wenn sich der Mauszeiger links und/oder oberhalb des Windows (bzw. des CLIP-Nullpunkts) befindet.

Beispiel: Im Zusammenhang mit der Maus stellt sich immer wieder das Problem, einen Bildschirmbereich zu umrahmen, um ihn für spätere Zwecke zu definieren oder zu kennzeichnen. Aus diesem Grund habe ich eine Prozedur entworfen, die Ihnen weitgehend freie Hand läßt und die nach eigenem Bedarf verändert oder erweitert werden kann. Diese Routine stellt allerdings DEFLINE 0,0,0,0, COLOR 1 und GRAPHMODE 1 ein, so daß ggf. nach Aufruf die vorher gültigen Einstellungen restauriert werden müssen. Sie können diese Einstellungen natürlich auch in der Routine verändern, um andere Effekte zu erzielen (z.B. in GRAPHMODE 3 oder COLOR 0).

Grundsätzlich funktioniert die Prozedur exakt genauso wie die Workbench-Funktion. Hier ist es allerdings gleichgültig, ob die linke, die rechte oder beide Maustasten gedrückt wurden. Ebenso wie die Workbench-Funktion muß die Prozedur mit gedrückter Maustaste aufgerufen werden, da sie sonst sofort wieder beendet wird. Ein wesentlicher Unterschied ist, daß diese Prozedur auch negative Minimalwerte verarbeitet, das heißt die Koordinaten der Endposition des Mauspfeils können kleiner sein als die Koordinaten der Startposition. In diesem Fall erhält man in den beiden Rückgabevariablen auch negative Box-Ausmaße.

```
Deffill ,2,4               ! DEFFILL grau
Pbox 10,10,200,200         ! Hintergrund zeichnen
Do                         ! Endlos-Schleife
  If Mousek                ! Maustaste gedrückt?
    Mouse X%,Y%,K%         ! Startkoordinaten holen
    Rubberbox(X%,Y%,-30,-30,*Xx%,*Yy%) ! Aufruf
    Box X%,Y%,X%+Xx%,Y%+Yy%! Box zeichnen
  Endif
Loop
'
```

```
Procedure Rubberbox(Xp%,Yp%,Xmin%,Ymin%,Xret%,Yret%)
  ' Xp%   = X-Startkoordinate
  ' Yp%   = Y-Startkoordinate
  ' Xmin% = Minimale Breite der Gummi-Box (auch neg.)
  ' Ymin% = Minimale Höhe der Gummi-Box (auch neg.)
  ' Xret% = Pointer auf 4-Byte-Integer, die nach Abschluß
  '         die letzte Breite der Box enthält (auch neg.)
  ' Yret% = Pointer auf 4-Byte-Integer, die nach Abschluß
  '         die letzte Höhe der Box enthält (auch neg.)
  Local Sc1$,Sc2$,Sc3$,Sc4$,Mx1%,Mx2%,Mx2%,My2%,Mk%
  Defline 0,0,0,0           ! DEFLINE voll/dünn
  Color 1                   ! COLOR schwarz
  Graphmode 1               ! Replace-Modus
  Repeat
    Mouse Mx1%,My1%,Mk%     ! Neue Koordinaten holen
    Mx2%=Max(Xp%+Xmin%,Mx1%) ! X-Koordinate auf Minimum trimmen
    My2%=Max(Yp%+Ymin%,My1%) ! Y-Koordinate auf Minimum trimmen
    Get Min(xp%,Mx2%),Min(yp%,My2%),...   !--. Es wird der
    ...Max(Mx2%,xp%),Min(yp%,My2%),Sc1$      ! Hintergrund
    Get Max(xp%,Mx2%),Min(yp%,My2%),...      ! jeder Box-Linie
    ...Max(Mx2%,xp%),Max(yp%,My2%),Sc2$      ! einzeln
    Get Min(xp%,Mx2%),Max(yp%,My2%),...      ! gesichert
    ...Max(Mx2%,xp%),Max(yp%,My2%),Sc3$      ! (Speicherplatz-
    Get Min(xp%,Mx2%),Min(yp%,My2%),...      ! Ersparnis!)
    ...Min(Mx2%,xp%),Max(yp%,My2%),Sc4$      !
    Box Xp%,Yp%,Mx2%,My2%                 !--'
    Repeat                                ! Warte...
      On menu                             ! Ereignis-Überwachung !
    Until Mousex<>Mx1% Or Mousey<>My1% Or Mousek=0
    ' ...bis die Maus bewegt oder ein Mausknopf gedrückt wird
    Put Min(Xp%,Mx2%),Min(Yp%,My2%),Sc4$  !--.
    Put Min(Xp%,Mx2%),Max(Yp%,My2%),Sc3$     ! Box-Hintergrund
    Put Max(Xp%,Mx2%),Min(Yp%,My2%),Sc2$     ! restaurieren
    Put Min(Xp%,Mx2%),Min(Yp%,My2%),Sc1$  !--'
  Until Mk%=0               ! Exit, wenn Maustaste = 0
  *Xret%=Mx2%-Xp%           ! Letzte Breite der Box zurück
  *Yret%=My2%-Yp%           ! Letzte Höhe der Box zurück
Return
```

Weitere Beispiele zur Maus-Abfrage finden Sie in Hülle und Fülle im Buch verteilt.

MOUSEX, MOUSEY, MOUSEK — Maus-Status ermitteln (einzeln)

```
Var=MOUSEX => X-Position
Var=MOUSEY => Y-Position
Var=MOUSEK => Maustastenstatus (siehe MOUSE)
```

Einzelabfrage-Funktionen für den jeweils gewünschten Maus-Status. In Version V3.0 siehe Anmerkung zu MOUSE.

STICK { STI } — Maus-Port-Abfragemodus bestimmen

`STICK(Modus) - Befehl -` *Version 3.0*

Bestimmt den Abfragemodus der Maus-/Joystick-Ports. Allerdings wurde er im Amiga-GFA-BASIC nur aus Kompatibilitätsgründen implementiert und hat selbst keine Wirkung.

Modus:

- 0 Mausmodus
- 1 Joystick-Modus

MOUSE-Befehle und STICK() schalten automatisch den entsprechenden Modus ein, wodurch der Befehl STICK in den meisten Fällen überflüssig ist.

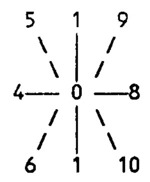
Ein Beispiel finden Sie unten unter STICK().

STICK() Maus-Port im Joystick-Modus abfragen

`Var=STICK(Maus-Port) - Funktion -` *Version 3.0*

Ermittelt den aktuellen Status des angegebenen Maus-Ports (0 oder 1). STICK(1) kann auch abgefragt werden, wenn durch den STICK-Befehl nicht der Joystick-Modus eingeschaltet ist. STICK(0) liefert dagegen nur im Joystick-Modus brauchbare Werte.

Es werden folgende Werte geliefert:



Als Bit-Vektor betrachtet sind von STICK() nur die untersten 4 Bits interessant:

Bit 0	Joystick hoch
Bit 1	Joystick runter
Bit 2	Joystick links
Bit 3	Joystick rechts

Beispiel: Dieses Programm erwartet, daß ein Joystick in Port 0 (eigentlich Maus-Port) angeschlossen ist.

```
CIRCLE 7,7,6          ! --.
CIRCLE 7,7,4              !
LINE 1,7,13,7             !- Pseudo-Sprite bauen
LINE 7,1,7,13             !
GET 0,0,15,15,a$      !---'
x%=160                ! X-Startkoordinate
y%=100                ! Y-Startkoordinate
STICK 1               ! Joystick-Modus einschalten
DO
  PRINT AT(1,1);"  ";AT(1,2);"  "
  IF STICK(0) AND 1   ! Joystick nach oben?
    y%=MAX(0,y%-1)    ! Y-Koordinate vermindern
```

```
  ENDIF
  IF STICK(0) AND 2  ! Joystick nach unten?
    y%=MIN(199,y%+1) ! Y-Koordinate erhöhen
  ENDIF
  IF STICK(0) AND 4  ! Joystick nach links?
    x%=MAX(0,x%-1)   ! X-Koordinate vermindern
  ENDIF
  IF STICK(0) AND 8  ! Joystick nach rechts?
    x%=MIN(319,x%+1) ! X-Koordinate erhöhen
  ENDIF
  VSYNC
  IF STRIG(0)        ! Fire-Button gedrückt?
    PRINT AT(1,1);"Feuer"
    COLOR 0
    FOR i%=0 TO 6    ! ———————.
      CIRCLE x%+7,y%+7,i%     !- Grafischer Effekt
    NEXT i%          ! ———————'
    PRINT AT(1,1);"     "
  ENDIF
  PUT x%,y%,a$       ! Pseudo-Sprite setzen
LOOP
```

STRIG() — Joystick-Fire-Buttons abfragen

`Var=STRIG(Maus-Port)` *Version 3.0*

Ermittelt den aktuellen Status des Fire-Buttons am angegebenen Maus-Port (gedrückt = 1, nicht gedrückt = 0).

Ein Beispiel finden Sie oben unter STICK().

14. Window- und Screen-Programmierung

Der Amiga ist in der Lage, beliebig viele Windows zu verwalten. Das steht in einem großen Gegensatz zum Atari ST, auf dem die Programmiersprache GFA-BASIC ihren großen Siegeszug begann. Trotzdem sind viele Befehle so übernommen worden. Sie haben einige zusätzliche Parameter bekommen. Außerdem hat der Amiga die Fähigkeit, diese vielen Windows auch noch auf unterschiedlichen Screens mit ganz verschiedenen Grafikeigenschaften zu verwalten. Auch dazu bietet die GFA-BASIC-Version 3.0 auf dem Amiga einige Befehle.

Zur Unterscheidung der Window- und Screen-Kommandos, die größtenteils mit gleichen Namen zu besetzen sind (siehe OPEN, CLOSE, MOVE ...), hat man sich überlegt, daß ein Buchstabe am Ende des Befehls ausreichen müßte, um den Unterschied deutlich zu machen. Deshalb enden alle Window-Befehle mit einem "W" und alle Screen-Kommandos mit einem "S".

14.1 Die Window-Befehle des GFA-BASIC

CLEARW { CL W } — Fenster-Inhalt löschen

```
CLEARW Nummer
CLEARW [#]Nummer (nur V3.0)
```

Das Fenster, dessen Nummer (0 - 15) hiermit übergeben wird, wird gelöscht. Der Befehl CLS kann ebenfalls verwendet werden. Allerdings kann dadurch nur das jeweils aktuelle Window gelöscht werden, und dieser Befehl kann Grafiken, die im rechten oder unteren Rand stehen, nicht erfassen.

Das Löschen des Fensterinhaltes ist immer dann notwendig, wenn das Fenster bewegt oder verkleinert wurde. Sie werden

feststellen, daß bei Bewegungen oder Größenänderungen der Fenster die Inhalte der übrigen Fenster überzeichnet werden. Die Verwaltung der jeweiligen Fensterinhalte ist allein Ihre Aufgabe, da niemand außer Ihnen wissen kann, was mit den Inhalten bei bestimmten Zuständen geschehen soll. Es gibt allerdings eine von Intuition angebotene Window-Verwaltung, die zusätzlichen Speicherplatz kostet. Lesen Sie dazu unter OPENW nach!

PRINT- sowie sonstige Text-Ausgaben können leicht durch eine Schleife restauriert werden, die die Textinhalte des Fensters neu in das Fenster hineinschreibt. Dagegen ist die Restauration von Grafik-Inhalten allerdings eine recht komplizierte Angelegenheit. In diesem Fall sollte man nicht auf die Hilfe von Intuition verzichten.

CLOSEW { CL W } — Fenster schließen

```
CLOSEW Nummer
CLOSEW [#]Nummer (nur V3.0)
```

Das Fenster mit der angegebenen Window-Nummer wird geschlossen. Dabei sind Werte zwischen 0 und 15 zugelassen.

FULLW { FUW } — Fenster auf maximale Größe bringen

```
FULLW Nummer
FULLW [#]Nummer (nur V3.0)
```

Nummer bestimmt die Nummer des Windows (0 - 15), das an die oberste Screen-Zeile gesetzt und bis zur untersten vergrößert werden soll. Auch die Breite des Windows wird auf den Maximalwert gebracht.

OPENW { O W } — Fenster öffnen

```
OPENW Nummer
OPENW Nummer, Xpos, Ypos, Breite, Höhe, IDCMP, Flaggen [, ScreenNr[,
BitMap]]
```

OPENW 'Nummer' öffnet bzw. aktiviert das Window, dessen Nummer (0 - 15) übergeben wird. Die Fenster sind dabei zuerst ohne Überlappungen folgendermaßen auf dem Workbench-Screen angeordnet:

Window 0: Füllt den gesamtem Screen abzüglich der Screen-Titelleiste aus.

Window 1 - 4:

Window 1	Window 2
Bildschirmmitte	
Window 3	Window 4

Window 5 - 15: Sind in der gleichen Größe wie Window 1 - 4, allerdings zentriert in der Screen-Mitte.

Die zweite Syntax-Variante gilt für V3.0 und ermöglicht es, über einen einzigen Befehl das gesamte Fenster zu definieren. Dadurch kann die umständliche Vorgehensweise, die bei Betriebssystem-Programmierung nötig ist, entfallen. Mit Hilfe der Positionsangaben kann man die linke obere Ecke des Windows auf dem Screen bestimmen. Dazu kommt noch die Höhe und Breite des Windows in Grafik-Pixeln der aktuellen Screen-Auflösung. Wichtig, weil unbedingt von Intuition benötigt, sind die nächsten zwei Parameter, die über die Eigenschaften des Windows Auskunft geben. Der erste Wert IDCMP steht für die Kommunikation des Windows mit dem Benutzer. Er gibt an, welche Nachrichten vom Window an das Programm gesendet

werden können, um so z.B. die Mitteilung über eine Veränderung der Größe loszuschicken. Dazu steht in direkter Abhängigkeit der zweite Wert, mit dem man speziell die Eigenschaften des Windows charakterisiert. Es ist hier möglich, aus den System-Gadgets bestimmte herauszuwählen oder aber etwas über die Verwaltung des Grafik-Speichers dieses Windows anzugeben.

Die nachfolgenden beiden Tabellen geben Auskunft darüber, welche Werte bei den beiden Parametern welche Wirkung haben. Da diese Werte aus der Betriebssystem-Programmierung stammen, hat man sie als Binär-Flags konstruiert. Das heißt, wir haben es bei jeder Eigenschaft mit einem korrespondierenden Bit-Flag zu tun, was im Dezimalsystem ungewöhnliche Werte erzeugt. Deshalb ist die Tabelle mit drei Werte-Spalten ausgerüstet. Die erste Spalte gibt den Wert in dezimal, die zweite in hexadezimal und die dritte in binär an, so daß es Ihnen überlassen bleibt, ein passendes Zahlensystem auszuwählen.

IDCMP-Einstellungen

dezimal	hexadezimal	binär	Flag-Name
1	&H00000001	&X00000000000000001	SIZEVERIFY
2	&H00000002	&X00000000000000010	NEWSIZE
4	&H00000004	&X00000000000000100	REFRESHWINDOW
8	&H00000008	&X00000000000001000	MOUSEBUTTONS
16	&H00000010	&X00000000000010000	MOUSEMOVE
32	&H00000020	&X00000000000100000	GADGETDOWN
64	&H00000040	&X00000000001000000	GADGETUP
128	&H00000080	&X00000000010000000	REQSET
256	&H00000100	&X00000000100000000	MENUPICK
512	&H00000200	&X00000001000000000	CLOSEWINDOW
1024	&H00000400	&X00000010000000000	RAWKEY
4096	&H00001000	&X00001000000000000	REQCLEAR
16384	&H00004000	&X00100000000000000	NEWPREFS
32768	&H00008000	&X01000000000000000	DISKINSERTED
65536	&H00100000	&X10000000000000000	DISKREMOVED
262144	&H00400000	&X00000000000000000	ACTIVEWINDOW
524288	&H00800000	&X00000000000000000	INACTIVEWINDOW
1048576	&H00100000	&X00000000000000000	DELTAMOVE
2097152	&H00200000	&X00000000000000000	VANILLAKEY
4194304	&H00400000	&X00000000000000000	INTUITICKS

Flaggen-Einstellungen

dezimal	hexadezimal	binär	Flags-Name
0	&H00000000	&X000000000000000000	SMARTREFRESH
1	&H00000001	&X000000000000000001	WINDOWSIZING
2	&H00000002	&X000000000000000010	WINDOWDRAG
4	&H00000004	&X000000000000000100	WINDOWDEPTH
8	&H00000008	&X000000000000001000	WINDOWCLOSE
16	&H00000010	&X000000000000010000	SIZEBRIGHT
32	&H00000020	&X000000000000100000	SIZEBOTTOM
64	&H00000040	&X000000000001000000	SIMPLEREFRESH
128	&H00000080	&X000000000010000000	SUPERBITMAP
256	&H00000100	&X000000000100000000	BACKDROP
512	&H00000200	&X000000001000000000	REPORTMOUSE
1024	&H00000400	&X000000010000000000	GIMMEZEROZERO
2048	&H00000800	&X000000100000000000	BORDERLESS
4096	&H00001000	&X000001000000000000	ACTIVATE
65536	&H00010000	&X010000000000000000	RMBTRAP
131072	&H00020000	&X100000000000000000	NOCAREREFRESH

Beispiel

Um z.B. ein Window zu öffnen, bei dem alle System-Gadgets vorhanden sind und das aktiv dargestellt wird, muß folgende Zeile eingegeben werden:

```
OPENW 0,0,0,640,256,0,4111
```

Zur Anwendung des IDCMP-Flags und zur Nachrichten-Übermittlung können Sie bei den Menü-Befehlen mehr erfahren.

TITLEW { TI } Fenster-Titelzeile bestimmen

```
TITLEW Nummer, "Window-Titel"[, "Screen-Titel"]
TITLEW [#]Nummer, "Window-Titel"[, "Screen-Titel"]
```

Das Fenster mit der angegebenen Nummer erhält im Titel-Balken diese neue Überschrift. Damit kann der Standard-Titeltext

"GFA-BASIC" ersetzt werden. Weiterhin ist es damit möglich, den Screen-Titeltext zu verändern. Und zwar wird dazu einfach ein weiterer Text angegeben, der dann bei Aktivierung in der Screen-Leiste erscheint. Dort steht sonst auf dem Workbench-Screen "Workbench Screen" zur Information an den Benutzer. Sie können diese Zeile z.B. für Programm-Infos oder andere Angaben benutzen.

FRONTW { FR } — Fenster in den Vordergrund

```
FRONTW Nummer
```

Version 3.0

Mit dieser Funktion wird das angegebene Window, das sich vielleicht im Hintergrund befindet, in den Vordergrund gebracht. Das ist wichtig, wenn eine aktuelle Ein- oder Ausgabe dort vorgenommen werden soll. Aber auch bei Informations-Fenstern, die vom Benutzer gesteuert werden können, geht man so sicher, daß sie im Vordergrund liegen.

Dieser Befehl ist besonders dann wichtig, wenn es keine Depth-Arrangement-Gadgets im Window gibt, mit denen der Benutzer die Lage bestimmen kann.

BACKW { BA } — Fenster in den Hintergrund

```
BACKW Nummer
```

Version 3.0

Dieser Befehl legt das über Nummer identifizierte Fenster in den Hintergrund. Damit gibt man Teile des Screens frei, die für andere Fenster gebraucht werden.

Dieser Befehl ist besonders dann wichtig, wenn es keine Depth-Arrangement-Gadgets im Window gibt, mit denen der Benutzer die Lage bestimmen kann.

MOVEW { MOV } — Fenster verschieben

`MOVEW Nummer, Xpos, Ypos` *Version 3.0*

Das Fenster mit der angegebenen Window-Nummer wird an die neue Position innerhalb des Screens bewegt. Damit kann die Benutzer-Operation, das "Draggen", auch vom Programm nachvollzogen werden.

Allerdings ist darauf zu achten, daß an der neuen Position auch noch die aktuelle Fenster-Größe möglich ist. Ist dies nicht der Fall, "schmiert" der Amiga hoffnungslos ab!

SIZEW { SIZ } — Fenstergröße bestimmen

`SIZEW Nummer, Breite, Höhe` *Version 3.0*

Die Größe eines Fensters läßt sich bei vorhandenem Sizing-Gadget vom Benutzer selbst einstellen. In einigen Fällen sollte aber diese Beeinflussung nur dem Programm überlassen sein, das dies mit wesentlich mehr Kompetenz durchführen kann (man weiß ja nie, was nicht alles vom Benutzer zerstört werden könnte).

Der Befehl SIZEW erlaubt nun die Angabe neuer Breiten- und Höhen-Maße für das betreffende Window, das mit der Nummer identifiziert wird.

Auch bei diesem Befehl ist darauf zu achten, daß die Angabe eines Wertes, der es erzwingt, das Window außerhalb des aktuellen Screens abzubilden, zu einem unbearbeiteten Absturz führt. Überprüfen Sie also zuerst, ob eine Vergrößerung in gewolltem Maße zulässig ist!

LIMITW { LIM } Fenstergröße einschränken

`LIMITW Nummer, Xmin, Ymin, Xmax, Ymax` *Version 3.0*

Selbst wenn man dem Benutzer mit dem Sizing-Gadget gestattet hat, die Größe des Windows einzustellen, muß dies trotzdem nicht in vollem Screen-Umfang geschehen. Es reicht durchaus, wenn man z.B. nur in einem Bereich von 100 - 400 Bildpunkten den X-Wert variieren kann.

Dazu gibt es das LIMITW-Kommando, das unter Angabe der Window-Nummer diesem neue Maximal- und Minimal-Werte zuweist. Intuition, also die Verwaltungsroutinen für alle Windows, Screens und andere Bedienelemente, kümmert sich dann selbst darum, daß keine anderen Werte gewählt werden.

SETWPEN { SETWP } Zeichenstifte festlegen

`SETWPEN LinienStift, FlächenStift` *Version 3.0*

Der Rahmen eines jeden Window ist uns von der Workbench her in ganz einheitlichen Farben bekannt. Die Linien werden in Weiß und der Korpus in Blau dargestellt. Das ist aber nicht unbedingt festgelegt. Wir können durchaus unter den vorhandenen Farben auswählen und dafür andere verwenden.

SETWPEN gibt mit seinen beiden Parametern neue Standard-Werte an. Das heißt, man muß diesen Befehl VOR dem Öffnen eines Fensters verwenden, um so die neue Farbgestaltung sichtbar zu machen. Es ist leider nicht möglich, nachträglich die Farben zu ändern.

WINDOW() { WINDOW() } **Window-Daten holen**

```
WINDOW(Nummer)
WINDOW(Adresse)
```

Version 3.0

Unter GFA-BASIC verwalten wir die Windows mit Nummern. Es ist viel einfacher, alle Funktionen mit einer Nummer aufzurufen, als sich die Position im Speicher zu merken, unter der die Window-Daten abgespeichert sind. Allerdings bleibt es nicht aus, daß man auch diese Adresse erfahren muß, um z.B. eine Betriebssystem-interne Funktion aufzurufen, die nur mit der Adresse arbeitet.

Mit WINDOW(Nummer) können wir einer Variablen die Adresse zuweisen, unter der die Window-Daten im Speicher stehen.

Umgekehrt läßt sich auch die Nummer eines Windows ermitteln, das von GFA-BASIC geöffnet wurde!

14.2 Die Screen-Befehle des GFA-BASIC

OPENS { OPENS } **Screen öffnen**

```
OPENS Nummer [, Xpos, Ypos, Breite, Höhe, BitPlanes, Modus]
```

Version 3.0

Dieser Befehl öffnet einen neuen Screen. Dafür können Indizes von 0 bis 15 (wie auch bei den Windows) verwendet werden. Voreingestellt ist unter der Nummer 0 der Workbench-Screen zu verstehen, auf der auch alle Ausgabe-Windows geöffnet werden, wenn man nichts anderes angibt.

Beim Öffnen eines neuen Screens muß man einige Parameter angeben, die die Eigenschaften des neuen Screens bestimmen. Wir haben es dabei quasi mit einem neuen Bildschirm zu tun, der genauso wie der Workbench-Screen gehandhabt werden kann.

Dabei kann die X- und Y-Position des Screens auf dem wirklichen Bildschirm bestimmt werden. Allerdings interessiert Intuition momentan nur der Y-Wert, weil bei dem jetzigen Grafik-Chip eine X-Verschiebung noch nicht möglich ist. Aus Kompatibilitätsgründen wurde aber der X-Wert implementiert. Sie sollten ihn immer auf 0 setzen, damit später keine Überraschungen zu erwarten sind.

Weiterhin ist die Breite und die Höhe für den Screen sehr wichtig. Sie entscheiden, wie viele Punkte maximal abgebildet werden können. Abhängig vom Grafikspeicher kann ein Screen bis zu 1024 x 1024 Bildpunkte groß sein, wenn genügend Grafikspeicher zur Verfügung steht (man benötigt schon für eine Bit-Plane 131072 Bytes, was 128 KByte entspricht). Es hängt dann allerdings vom Grafikmodus ab, wie viele dieser Punkte des Screens wirklich innerhalb des Ausschnitts dargestellt werden können, der auf dem Bildschirm sichtbar ist. Dieser Bereich beschränkt sich nämlich auf 640 x 256 Punkte, von denen man noch einige wenige abrechnen kann, die durch den Rand verloren gehen, in dem aber sowieso keine scharfe Abbildung möglich ist (das liegt aber nicht am Amiga, sondern am Monitor).

Zu den Auflösungen sehen Sie am Ende der Beschreibung eine Tabelle, aus der Sie die Werte entnehmen können. So wichtig wie die Auflösung ist auch die Anzahl der BitPlanes. Oben wurde schon darauf verwiesen, daß jede BitPlane Speicher in Anspruch nimmt. In diesem Speicher werden die gesetzten oder nicht gesetzten Punkte der Grafik verwaltet. Jedem Punkt ist ein Bit zugeordnet. Bei gesetztem Bit sehen wir den Punkt, und bei gelöschtem Bit sieht man nur die Hintergrundfarbe.

Zur Darstellung von mehr Farben als nur diesen beiden kombiniert man mehrere BitPlanes und definiert einen Code, unter dem sich die richtige Farbe berechnen läßt. Allgemein steht die Anzahl der Farben in folgender Abhängigkeit zu den BitPlanes:

```
Farben = 2^BitPlanes
```

Dabei gilt diese Tabelle:

BitPlanes	Farben
0	1 (die Hintergrundfarbe und mehr nicht!)
1	2
2	4
3	8
4	16
5	32

Der Amiga ist nicht in der Lage, mehr als 6 BitPlanes zu verarbeiten. Obwohl man alle 4096 verschiedenen Farben darstellen kann, wird dabei immer noch mit 6 BitPlanes gearbeitet. Allerdings ist das Verfahren zur Farbkodierung dann anders gewählt.

Hier kommt als Abschluß noch die Tabelle der Screen-Modi und eine Auflösungstabelle, anhand derer Sie ablesen können, wieviele Grafikpunkte auf dem Bildschirm bei welchem Modus zu sehen sind:

Die Screen-Modi:

dezimal	hexadezimal	Modus
2	$0002	GENLOCK_VIDEO
4	$0004	INTERLACE
128	$0080	EXTRA_HALFBRITE
2048	$0800	HOLD_AND_MODIFY
16384	$4000	SPRITES
32768	$8000	HIRES

Die verschiedenen Auflösungsstufen:

Bezeichnung	hexadezimal	Auflösung max.	Bitplanes
LORES	$0000	320 x 256	5
HIRES	$8000	640 x 256	4
INTERLACE	$0004	320 x 512	5
HIRES INTERLACE	$8004	640 x 512	4
EXTRA_HALFBRITE	$0080	320 x 256	6
EXTRA_HALFBRITE LACE	$0084	320 x 512	6
HOLD_AND_MODIFY	$0800	320 x 256	6
HOLD_AND_MODIFY	$0804	320 x 512	6

CLOSES { CLOSES } — Screen schließen

`CLOSES Nummer` *Version 3.0*

Mit diesem einfachen Kommando wird ein geöffneter Screen wieder geschlossen. Dabei werden auch alle auf dem Screen befindlichen Windows gleich mit geschlossen, so daß man sich darum nicht kümmern muß.

FRONTS { FRONTS } — Screen in den Vordergrund

`FRONTS Nummer` *Version 3.0*

FRONTS holt den mit Nummer angegebenen Screen in den Vordergrund. Dabei wird die gleiche Funktion ausgeführt, wie es auch der Fall ist, wenn man mit der Maus das entsprechende Depth-Arrangement-Gadget des Screens betätigt. Nur hier erfolgt die Kontrolle über das Programm.

BACKS { BACKS } — Screen in den Hintergrund

```
BACKS Nummer
```

Version 3.0

BACKS legt den über die Nummer identifizierten Screen in den Hintergrund, so daß der zweitoberste Screen sichtbar wird. Dabei wird die gleiche Funktion ausgeführt, wie es auch der Fall ist, wenn man mit der Maus das entsprechende Depth-Arrangement-Gadget des Screens betätigt. Nur hier erfolgt die Kontrolle über das Programm.

MOVES { MOVES } — Screen verschieben

```
MOVES Nummer, Xpos, Ypos
```

Version 3.0

Bewegt den angegebenen Screen zur absoluten X-, Y-Position. Man kann damit einen im Hintergrund liegenden Screen teilweise sichtbar machen, und so unterschiedliche Auflösungen gleichzeitig auf dem Bildschirm darstellen. Das eignet sich besonders für gemischte Text- und Grafik-Ausgabe.

TITLES { TITLES } — Screen-Titel setzen

```
TITLES Nummer, "Titel"
```

Version 3.0

Wie auch jedes Window einen Titel in seiner Kopf-Zeile beinhaltet, so haben auch Screens die Möglichkeit, in der Titel-Zeile einen Text aufzunehmen.

Mit diesem Befehl wird der Text definiert, der in der obersten Zeile des Screens mit der angegebenen Nummer steht. Diesen Titel bezeichnet man als Default-Title, weil er nur dann angezeigt wird, wenn kein Fenster innerhalb der Screen aktiv ist. Je-

des Fenster hat nämlich zusätzlich zum Fenster-Titel auch noch einen Text, der bei Aktivierung in der Screen-Zeile ausgegeben werden soll. Nur wenn dieser zweite Text fehlt, benutzt der Screen den Default-Title.

SETSPEN { SETSPEN } — Screen-Farben wählen

```
SETSPEN LinienStift, FlächenStift
```

Version 3.0

Genau wie auch bei den Windows werden zum Zeichnen der Screen-Elemente wie Titelleiste und Gadgets zwei Farbstifte benutzt. Diese sind auf die Werte 1 und 0 eingestellt, können aber beliebig (je nach Auflösung) gewählt werden. Dazu dient dieser Befehl, der VOR dem Öffnen des Screens aufgerufen werden muß, da sonst die Einstellungen nicht bekannt sind.

SCREEN() { SCREEN() } — Screen-Daten holen

```
SCREEN(Nummer)
SCREEN(Adresse)
```

Version 3.0

Für die interne Programmierung der Intuition-Elemente ist es notwendig, daß man die Adresse kennt, ab der die Screen-Daten abgespeichert werden. Diese läßt sich mit der ersten Funktion ermitteln, die die entsprechende Speicherstelle zurückliefert.

Kennt man nur die Speicherstelle und benötigt für die GFA-Befehle die Screen- Nummer, so kann diese mit der zweiten Variante der Funktion ermittelt werden.

15. Menüprogrammierung mit BASIC-Befehlen

MENU Menü — Menüpunkt-Attribute bestimmen

```
MENU Menüpunkt,Attribut
```

Mit diesem Befehl kann bestimmt werden, ob ein Menüpunkt ghosted (inaktiv) dargestellt werden soll oder klar (aktiv). Außerdem können vor einem Menüpunkt-Text Häkchen (Checkmarks) gesetzt werden. Diese kennzeichnen üblicherweise, daß die Funktion, die durch den gewählten Menüpunkt repräsentiert wird, z.Zt. aktiv ist.

Dazu wird dem Befehl in Menüpunkt der Index des zu markierenden bzw. des zu desaktivierenden Menüpunktes sowie nach einem Komma das gewünschte Attribut übergeben.

Attribut:

64	Checkmark löschen
256	Checkmark setzen (abhaken)
16	Menüpunkt desaktivieren (hell)
64	Menüpunkt aktivieren

Bei MENU Menütext$ sind für den Fall, daß Checkmarks (Häkchen) gesetzt werden sollen, vor der jeweiligen Menüpunkt-Bezeichnung zwei Leerzeichen vorzusehen.

MENU Text — Menüpunkt mit neuem Text versehen

```
MENU Menüpunkt,Text
```

Sollte es der Fall sein, daß einer der schon über das Feld definierten Menüpunkte nicht mehr den richtigen Text besitzt, weil

Sie z.B. eine Funktion entfernt, erweitert oder verändert haben, dann ist diese Funktion genau die richtige!

Mit Menüpunkt geben Sie die Nummer des gewünschten Menüpunktes an, und in Text wird ein String übergeben, der als neuer Text gesetzt werden soll. Beachten Sie dabei, daß dieser neue Text auf keinen Fall länger sein darf, als es der erste war. Denn GFA-BASIC verwendet für den Menüpunkt die alte Text-Struktur und schreibt nur den neuen Text dort hinein. Deshalb ist kein zusätzlicher Speicher für längere Texte reserviert.

MENU KEY — Menüpunkt mit Shortcut versehen

```
MENU KEY Menüpunkt,ASCIIcode
```

Wie Sie wissen, kann jeder Menüpunkt auch über eine Tastenkombination aufgerufen werden. Zusammen mit der rechten Amiga-Taste kann jede andere Taste mit dem Aufruf eines Menüpunktes belegt werden. Dazu wird dem Befehl zur Nummer des Menüpunktes noch der ASCII-Wert der zu drückenden Taste übergeben. Zur Verdeutlichung dieser Möglichkeit für den Anwender erscheint dann im Menü die Amiga-Taste als Symbol zusammen mit dem Zeichen. Wenn Sie dies beabsichtigen, dann sollten Sie aber gleich beim Menütext vier oder mehr Zeichen freihalten, damit das Symbol zusammen mit dem Tastenwert nicht in den Menütext hineinragt, was einigen grafischen Müll erzeugt.

MENU Menütext$() — PullDown-Menü erstellen

```
MENU Array$()
```

Mit diesem Befehl wird dem Interpreter gesagt, wo er im Speicher die einzelnen Menüeinträge findet. Sie übergeben ihm dazu

den Namen eines eindimensionalen String-Arrays, das die Menütexte enthält. Dieses Array muß so viele Elemente aufweisen, wie insgesamt an Menüpunkt-Texten zugewiesen werden sollen. Hierbei ist zu beachten, daß das erste Menü (Desktop-Menü für Accessories) etwa folgenden Aufbau haben sollte (der sich an den ST-Standard anlehnt, aber deshalb nicht unbedingt schlecht ist):

1. String = Menütitel (evtl. Programmname).

2. String = Beliebige Überschrift (üblich: Programm-Info). Unter diesem Menüpunkt können Sie eine beliebige Programm-Funktion einordnen. Da das aber der einzige verwendbare Menüpunkt in diesem ersten Menü ist, wird er normalerweise für die Ausgabe einer allgemeinen Programm-Information (z.B. Copyright) verwendet.

3. String = Reihe von Minuszeichen (bzw. Bindestrichen). Die Anzahl der Striche bestimmt hier die Menübreite. Da in diesem Menü grundsätzlich die evtl. vorhandenen Desk-Accessories aufgeführt werden, sollten Sie hier die Länge des längsten Accessory-Titels berücksichtigen. Dieser Strich unterteilt ein Menü in verschiedene Funktionsblöcke.

4. String = ab hier folgen die einzelnen Funktionen des ersten Menüs, die allgemein mit Service-Aufgaben belegt sein sollten. Bis wir dann zu den letzten Menüpunkten kommen:

...

9.

10. String = Null-String. Dieser Null-String gilt als Abschluß für jede einzelne Menüreihe.

An diesen ersten Menüaufbau werden nun die von Ihnen frei benennbaren weiteren Menüs angehängt. Diese sind so aufgebaut, daß sich an den Menütitel (der String, der immer in der Menüleiste sichtbar bleibt) die einzelnen Menüpunkt-Bezeich-

nungen anschließen. Dabei muß auch hier, wie beim ersten Menü, ein Null-String ("") den Abschluß zu jedem einzelnen Menü bilden.

Nachdem alle Menüeinträge gelesen wurden, muß dem gesamten Menüaufbau noch ein Null-String als Endmarkierung angehängt werden. Wollen Sie, daß bei Anwahl eines Menüpunktes dieser mit einem Häkchen (Checkmark siehe MENU Menüpunkt,Attribut) versehen wird, sollten Sie vor dem Text des gewünschten Menüpunktes zwei Leerzeichen vorsehen, damit dieses Checkmark genügend Platz hat. Außerdem ist es möglich, einen Menüpunkt-Text ghosted (verschleiert = inaktiv) darzustellen. Dazu muß das erste Zeichen des Text-Strings ein Minuszeichen (bzw. Bindestrich: -) sein. Nachdem das Menü eingelesen wurde, wird es durch diesen Befehl gleichzeitig aktiviert.

Beispiel:

```
DIM array$(40)
REPEAT
  READ array$(i%)
  PRINT array$(i%)
  INC i%
UNTIL array$(i%-1)="XXX"
array$(i%-1)=""
DATA DESKTITEL,Menüpunkt
DATA -------------------------------
DATA Acc1,Acc2,Acc3,Acc4,Acc5,Acc6,
DATA MENÜ1,  Punkt1,  Punkt2,  Punkt3,
DATA MENÜ2,  Punkt1,  Punkt2,  Punkt3,
DATA MENÜ3,  Punkt1,  Punkt2,  Punkt3,
DATA MENÜ4,  Punkt1,  Punkt2,  Punkt3,
DATA XXX
MENU array$()
```

MENU KILL — Menüzeile löschen

```
MENU KILL
```

MENU KILL desaktiviert das PullDown-Menü. Es kann keine Auswahl mehr vorgenommen werden, weil die gesamte Menü-

leiste vom Window entfernt wurde. Auf den Inhalt des Menütext-Arrays hat dieser Befehl keinen Einfluß. Das desaktivierte Menü kann also jederzeit durch MENU Menütext$ wieder installiert werden.

16. Ereignis-Überwachung mit BASIC-Befehlen

MENU(Index) Event-Puffer (Menü- und Fensterverwaltung)

```
Var=MENU(Index)
```

Hinter dieser Funktion verbirgt sich ein Datenfeld (Event-Puffer = Integerfeld), in dem permanent verschiedene Daten zu aktuellen Intuition-Ereignissen eingetragen werden.

Index steht für das jeweils interessante Feldelement. Dabei kann der Bereich von MENU(0) bis MENU(10) abgefragt werden, in dem wir alle nötigen Daten zur Ereignis-Verwaltung finden. Sehen Sie in der ersten Tabelle die Funktion der jeweiligen Feld-Einträge, die dann noch speziell in Abhängigkeit von dem Nachrichtentyp betrachtet werden:

Die Inhalte von MENU(Index):

MENU(0) Menu — Nummer des ausgewählten Menüpunktes innerhalb des String-Arrays der Menüs, das bei der Definition übergeben wurde.

MENU(1) Class — Enthält die IDCMP-Flags, die diese Nachricht ausgelöst haben. Je nach Flaggen-Typ müssen die nächsten drei Felder unterschiedlich ausgewertet werden.

MENU(2) Code — Dieser Wert enthält in Abhängigkeit der Message-Class eine Zahl, die z.B. bei einer Menüauswahl die Nummer des ausgewählten Menüpunktes beinhaltet.

MENU(3) Qualifier In dieser Variablen sind die Qualifier, also die Sondertasten wie SHIFT, CTRL etc. vermerkt. Das ist bei der Abfrage der Tastatur von entscheidender Bedeutung.

MENU(4) IAddress Der hier abgelegte Wert ist ein Zeiger (eine Speicheradresse) auf das Intuition-Objekt, das für die Nachricht verantwortlich war. Das kann ein Gadget sein oder auch ein Window.

MENU(5) MouseX Egal, welche Nachricht empfangen wurde, in dieser Variablen ist die relative X-Position des Mauspfeils zu finden. Sie können auch damit die Mausposition auswerten!

MENU(6) MouseY Hier steht passend zu MENU(5) die Y-Position der Maus. Diese wird relativ zur oberen linken Ecke des Windows berechnet. Befindet sich das Window also nicht ganz in der oberen Ecke des Screens, kann es bei beiden Koordinaten-Teilen negative Werte geben.

MENU(7) Seconds So, wie die Position der Maus zum Zeitpunkt der ausgelösten Nachricht festgehalten wird, speichert Intuition, von dem diese Nachricht stammt, auch die System-Zeit, damit bei einer verzögerten Auswertung der Zeitpunkt genau bekannt ist.

MENU(8) Micros Im Gegensatz zu MENU(7) stehen hier nicht Sekunden, sondern Mikrosekunden.

MENU(9) WindowAddress

Hier steht immer ein Zeiger auf das Window, von dem die Nachricht stammt.

Ein Menü abfragen:

Um die Auswahl eines Menüs abzufragen, empfiehlt sich folgendes Vorgehen, das neben der hier besprochenen Funktion noch weitere GFA-BASIC-Befehle benutzt. Bitte lesen Sie auch dort die Informationen nach.

Im Hauptprogramm wird nach der Definition der Menüleiste ein Unterprogramm definiert, in dem die Auswertung stattfindet. Das sieht so aus:

```
ON MENU GOSUB Auswertung
```

Die Hauptprogramm-Schleife kann im einfachsten Fall sogar so aussehen:

```
REPEAT
  SLEEP
UNTIL Loop=FALSE
```

Damit wird solange auf eine Nachricht gewartet, bis eine globale Variable (hier Loop) von irgendeiner Unterroutine auf FALSE gesetzt wird. Sie können aber natürlich auch etwas mehr vom Programm abarbeiten lassen, bis die erste Menü-Nachricht eintrifft.

Die Unterroutine selbst kann mit Hilfe der MENU()-Funktion die jeweilige Nummer des augewählten Menüpunktes ermitteln und entsprechend zu einer weiteren Routine verzweigen, die eine gewünschte Aufgabe erfüllt oder Prozedur ausführt.

```
PROCEDUR Auswertung
  Menu%=MENU(0) ' Die Menü-Nummer wird der Variablen übergeben
  MENU KILL     ' damit keine weitere Auswahl für den Benutzer
möglich ist
  IF Menu%=1 THEN GOSUB Speichern
  IF Menu%=2 THEN GOSUB Laden
  ...
  MENU Menu.Leiste$()
RETURN
```

Es gibt natürlich noch andere und sehr verschiedene Nachrichten, die alle über die MENU()-Funktion abgefragt werden kön-

nen. So lassen sich alle IDCMP-Flags des Windows nutzen. Diese Flaggen geben Sie bei der Definition eines neuen Windows an. Dadurch sagen Sie dem verwaltenden Intuition-System, bei welchen Ereignissen Sie eine Information wünschen. Die Zahlen, die von MENU(1) dabei geliefert werden, entsprechen exakt denen, die auch als Flags bei der Window-Definition angegeben werden müssen.

Menü-Nachrichten:

Neben der einfachen Abfrage über MENU(0), wo sich der Menü-Index befindet, gibt es natürlich eine spezielle Intuition-Nachricht, die die Auswahl eines Menüpunktes genau beschreibt. Allerdings wird hier eine genauere Auswertung vom Benutzer verlangt.

MENU(1) = 256 — Es wurde ein Menüpunkt der bestehenden Menüleiste ausgewählt.

In MENU(2) steht die genaue Nummer des Menüpunktes in einem Bitmuster verschlüsselt. Sie können dabei genau auslesen, in welchem Menü welcher Menüpunkt und welcher Untermenüpunkt angewählt wurde. Verwenden Sie dazu die folgende Dekodierung: MenüNummer = MENU(2) AND 31, MenüPunkt = SHR(MENU(2), 5) AND 63, UnterMenüPunkt = SHR(MENU(2), 11) AND 31.

Window-Nachrichten:

MENU(1) = 1 — Das Size-Gadget des Window wurde betätigt.

In MENU(10) findet man jetzt die Intuition-Adresse des Fensters.

MENU(1) = 2

Die Größe des Fensters wurde verändert.

In MENU(10) steht auch hier wieder die Adresse der Window-Struktur (Sie kommen mit WINDOW(Adresse) an die GFA-BASIC-Nummer, unter der das Window verwaltet wird).

MENU(1) = 4

Eine Aufforderung an das Programm, den Grafik-Inhalt (auch Texte werden als Grafik behandelt) des Windows wieder herzustellen, weil er durch vorhergehende Überlagerung zerstört wurde.

In MENU(10) steht die Adresse des Windows.

MENU(1) = 512

Das Close-Gadget des Windows wurde betätigt.

In MENU(10) steht die Adresse der Window-Struktur.

MENU(1) = 8192

Das Fenster ist nicht in der Lage, die über die rechte Maustaste angeforderte Menüleiste auszugeben.

In MENU(10) steht die Adresse des Windows.

MENU(1) = 262144

Ein Fenster wurde über einen Mausklick aktiviert.

Die Adresse des Fensters steht in MENU(10) und kann mit WINDOW(Adresse) in die GFA-Nummer des Windows umgewandelt werden.

MENU(1) = 524288 Ein Fenster wurde desaktiviert, weil ein anderes aktiviert wurde oder der Benutzer mit der Maus auf den Screen geklickt hat.

Die Adresse des Fensters steht in MENU(10) und kann mit WINDOW(Adresse) in die GFA-Nummer des Windows umgewandelt werden.

Maus-Nachrichten:

MENU(1) = 8 Eine Maustaste wurde betätigt (Sie erhalten nur Nachrichten von der rechten Maustaste, wenn innerhalb dieses Fensters die Menü-Steuerung ausgeschaltet wurde und die rechte Maustaste wie die linke gehandhabt wird).

In MENU(2) steht der Status der Maustasten, aus dem sich auslesen läßt, welche Taste gedrückt oder losgelassen wurde. So steht der Wert 104 für das Drücken der linken Maustaste und die Zahl 232 für die rechte. Es ist jeweils 1 dazu zu addieren, wenn die Taste losgelassen wurde.

MENU(1) = 16 Die Maus wurde bewegt

In MENU(5) und MENU(6) lassen sich die absoluten X- und Y-Koordinaten relativ zum Fenster auslesen.

MENU(1) = 1048576 Die Maus wurde bewegt.

In MENU(5) und MENU(6) lassen sich die relativen X- und Y-Koordinatenveränderungen zur letzten Position innerhalb des Fensters auslesen.

Gadget-Nachrichten:

MENU(1) = 32 Ein Gadget wurde gedrückt.

In MENU(4) finden Sie den Zeiger auf die Gadget-Struktur (die Spei-cheradresse, ab der Gadget-Daten abgelegt sind) und in MENU(10) die Window-Adresse des Windows, in dem sich das Gadget befindet.

MENU(1) = 64 Ein Gadget wurde wieder losgelassen.

Auch hier steht in MENU(4) die Adresse der Gadget-Struktur und in MENU(10) die der Window-Struktur.

Requester-Nachrichten:

MENU(1) = 128 Innerhalb des Fensters ist ein Requester geöffnet worden.

MENU(4) enthält die Adresse der Requester-Struktur und MENU(10) die des betroffenen Windows.

MENU(1) = 2048 Es ist nicht möglich, den Requester im Fenster darzustellen.

In MENU(10) steht die Adresse des Fensters.

MENU(1) = 4096 Der letzte im Fenster geöffnete Requester wurde gerade geschlossen (das Window ist wieder zur Ein- und Ausgabe freigegeben).

In MENU(10) steht die Adresse des Fensters.

Tastatur-Nachrichten:

MENU(1) = 1024 Eine Taste wurde betätigt.

In MENU(2) ist der Tastatur-Code der gedrückten Taste abgelegt, zu dem noch der entsprechende Qualifier kommt, der den Status der Tastatur anzeigt. Er steht in MENU(3).

MENU(1) = 2097152 Eine Taste wurde betätigt.

In MENU(2) steht der ASCII-Code der betätigten Taste. Es erfolgte also schon eine Auswertung anhand der Tastaturtabelle.

Andere Nachrichten:

MENU(1) = 16384 Von irgendeinem anderen Programm wurden die Grundeinstellungen des Systems verändert (die Preferences-Daten sind neu eingestellt worden).

MENU(1) = 32768 In irgendein Laufwerk wurde eine Diskette neu eingelegt.

MENU(1) = 65536 Aus irgendeinem Laufwerk wurde die Diskette entfernt.

MENU(1) = 131072 Die Workbench schickt diese Nachricht, wenn auf ihr Veränderungen vorgenommen wurden.

MENU(1) = 4194303 Bei Anforderung von Zeit-Nachrichten erhält man über jeden IntuiTick eine Nachricht. Dies geschieht alle 1/10tel Sekunde!

ON MENU — Verzweigung zur Ereignisfeststellung

```
ON MENU
ON MENU [Zeit]
```

Dieser Befehl wartet darauf, daß eine Nachricht eingetroffen ist und verzweigt sogleich zu der definierten Stelle im Programm. Er sollte innerhalb einer Schleife durchlaufen werden, denn nur bei einer permanenten Abfrage ist es möglich, keine Nachricht zu übersehen. Wird der Befehl nicht eingesetzt, kann auch keine der ON MENU...GOSUB-Prozeduren angesprungen werden.

Es kann optional der Parameter Zeit eingesetzt werden. Dieser Wert gibt in 1000stel Sekunden an, nach welcher Zeitspanne ON MENU (siehe EVNT_MULTI) spätestens abgeschlossen werden soll und die Kontrolle wieder an BASIC zurückgegeben wird. Bei ausreichender Zeitvorgabe hat so das Input-Device Zeit, ggf. das Loslassen des Mausknopfes zu registrieren. Dazu ist allerdings unbedingt erforderlich, ON MENU BUTTON GOSUB einzusetzen, da sonst der Zeitwert unberücksichtigt bleibt.

Vergessen Sie nicht, diesen Befehl auch innerhalb von internen Warteschleifen anzuführen (z.B. REPEAT...UNTIL MOUSEK), da sonst solange kein Ereignis bearbeitet werden kann, wie sich das Programm in dieser Warteschleife befindet.

SLEEP — Auf eine Nachricht warten

```
SLEEP
```

Der SLEEP-Befehl ist eine Vereinfachung des ON- MENU-Kommandos. Er setzt einfach für die Zeit-Variable den Wert 1 ein und wartet auf irgendeine Nachricht, die über den Nachrichten-Kanal des Intuition geschickt wurde.

ON MENU GOSUB Procedure-Bestimmung (Menü-Event)

```
ON MENU GOSUB Prozedurname
```

Prozedurname gibt die Prozedur an, zu der verzweigt werden soll, wenn ein Pull-Down-Menüpunkt (Menüeintrag) angeklickt wurde. Über MENU(0) kann dann dort der gewählte Menüpunkt ermittelt und dementsprechend reagiert werden. Existiert Prozedurname nicht, wird die Menü-Überwachung abgeschaltet. Sollten Sie diesen Befehl aufrufen, bevor ein Fenster mit einem Menü definiert wurde, stürzt das Programm ab!

ON MENU BUTTON GOSUB Procedure-Bestimmung

```
ON MENU BUTTON GOSUB Prozedurname
```

(Mausknopf-Event) Durch diesen Befehl kann eine Prozedur bestimmt werden, zu der verzweigt werden soll, sobald eine oder mehrere Maustasten ein- oder mehrmals gedrückt oder auch nicht gedrückt wurden. Innerhalb der Routine muß dann eine Auswertung über die MENU()-Funktion geschehen, in der ja ausgelesen werden kann, welche Taste wann wie oft gedrückt wurde.

ON MENU KEY GOSUB Proc.-Bestimmung (Tastatur-Event)

```
ON MENU KEY GOSUB Prozedurname
```

Die Tastatur wird überwacht und bei eingetretenem Ereignis (ON MENU nicht vergessen) zu Prozedurname verzweigt, wo man dann durch MENU(14) den Code der gedrückten Taste erfahren kann. Existiert Prozedurname nicht, wird die Tastatur-Überwachung abgeschaltet.

17. Der GFA-Compiler

Compiler? Was ist ein Compiler?

Dieser Begriff sollte am besten durch einen Vergleich erklärt werden. Wenn Sie normalerweise mit GFA-BASIC arbeiten, werden Sie ein Programm in den Editor eingeben und dieses mit RUN starten. Danach macht sich GFA-BASIC an die Arbeit: Zeile für Zeile untersucht es die Befehle und führt sie nacheinander in der vorgegebenen Reihenfolge aus. Dieses Verfahren nennt man *interpretieren.* Das GFA-BASIC ist also ein *Interpreter.*

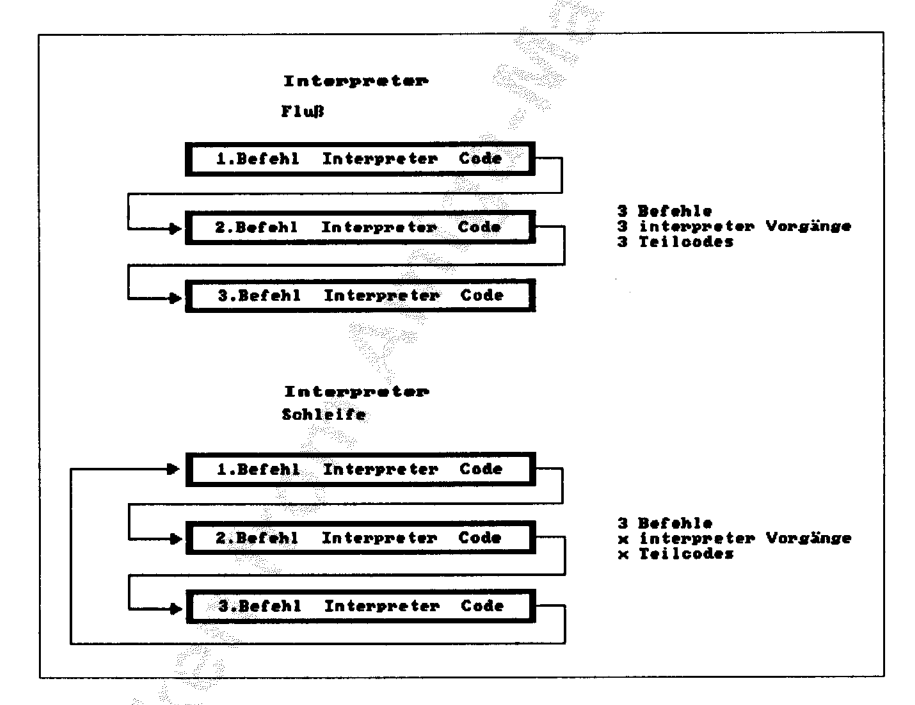


Abbildung 17.1/1: Arbeitsweise eines Interpreters

Man kann sich schon als Laie vorstellen, daß dieses Verfahren zwar - weil es sehr plausibel und einfach verständlich ist - auf der Hand liegt, jedoch in dieser Einfachheit auch einige Tücken liegen. Man kennt diese Technik aus dem Alltag. Jemand möchte einem anderen Aufgaben zuteilen und schreibt diese auf. Derjenige, für den diese Aufgaben gedacht sind, liest sie sich nacheinander durch und führt (je nach Gewissenhaftigkeit) ein nach der anderen aus. Das Problem beim Computer liegt nun darin begründet, daß er selber diese Anweisungen so nicht versteht. BASIC ist eine Hochsprache, d.h. sie ist für den Menschen verständlich, muß aber erst auf die tiefere Ebene der Computersprache (Maschinensprache) übersetzt werden. Dieser Übersetzungsvorgang nimmt, wie man es aus der Simultanübersetzung her kennt, einige Zeit in Anspruch. Je nach Güte des Übersetzers kann es schnell oder weniger zügig von Statten gehen.

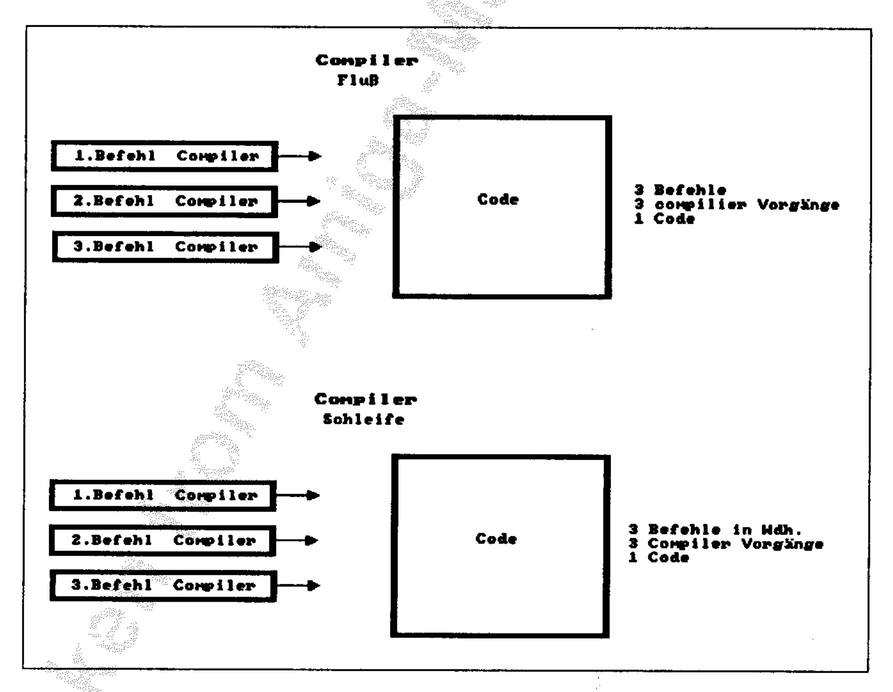


Abbildung 17.1/2: Arbeitsweise eines Compilers

Es gibt dafür eine grundsätzlich schnellere Methode. Man übersetzt die Sprache zuvor einmalig (so z.B. bei Computerbüchern) und kann sich dann bei der Ausführung die Übersetzung sparen. Diese Vorübersetzung nennt man *compilieren*, das passende Programm dazu *Compiler*.

Damit sind wir endlich bei unserem Thema, dem Compiler. Die obenstehende Grafik will noch einmal den Unterschied zwischen den beiden Techniken der Befehlsausführung verdeutlichen.

In der Praxis

Wie wirkt sich nun dieser recht deutliche Unterschied in der Verarbeitung bei der täglichen Arbeit aus? Würde nur eine Methode die deutlich bessere sein, gäbe des die andere nicht. Doch dies ist nicht der Fall. Man kann aber ohne ein schlechtes Gewissen gegenüber den Entwicklern haben zu müssen sagen, daß es um einiges einfacher ist, einen Interpreter zu entwickeln, als einen Compiler.

Trotzdem ist der Interpreter kein schlechtes Werkzeug. Besonders im Stadium der Entwicklung eines Programms, macht es sich oft viel einfacher, den Code, wenn auch langsamer interpretieren zu lassen, dafür aber schnell eine Veränderung durchführen zu können. Sie kennen das sicherlich aus eigener Erfahrung:

Man schreibt ein Programm, das mit der Zeit immer länger wird und damit eine solche Komplexität erreicht, daß es nicht mehr ohne weiteres überschaubar ist. Nun wird eine Änderung vorgenommen. Diese läßt sich auf ihre Funktionsfähigkeit ganz leicht dadurch überprüfen, daß man das Programm startet. Der Interpreter wird schon an der neuen Stelle einen Fehler melden, wenn einer vorhanden sein sollte.

Nicht so beim Compiler. Dieser benötigt für die Übersetzung zwar insgesamt nicht viel länger. Muß aber auch Programmteile übersetzen, die bisher schon vollkommen funktionsfähig waren. Man hat also einige Zeit zu warten. Tritt während der Übersetzung ein Syntax-Fehler auf, bricht der Compiler ab (meistens ohne eine Hilfestellung, wo der Fehler liegen mag). Während der

Interpreter also bei der Fehlersuche sehr hilfreich ist, verweigert der Compiler jede konstruktive Zusammenarbeit.

Ist ein Programm aber fertig entwickelt, und soll dieses dann unter besten Voraussetzungen laufen, empfielt es sich, dieses mit dem Compiler zu beschleunigen. Wenn das Programm ausreichend nach Fehlern untersucht wurde, sollten Syntax-Fehler nicht mehr vorliegen, die Compilierung kann ohne Probleme ablaufen. Als Ergebnis erhält man ein wesentlich schnelleres Programm.

Der Compiler auf dem Amiga

Nach dieser grauen Theorie zum Thema Compiler soll es jetzt an die Arbeit gehen. Der GFA-Compiler wird auf einer Diskette zusammen mit Hilfsdateien geliefert. Von dieser Diskette sollten Sie sich - ganz gleich, welche Computer-Konfiguration Sie haben - eine Kopie anfertigen.

Wir skizzieren diesen Weg noch einmal in kurzen Zügen:

Von der Workbench aus legen Sie die Original-Diskette in ein beliebiges Laufwerk. Sollten Sie nur eines besitzen, bleibt dafür natürlich nur DF0. Sie klicken nun das auf dem Workbench-Screen erschienene Symbol der Diskette einmal an und wählen im Menü Workbench den Punkt Duplicate aus. Nach einer Aufforderung die Workbench-Diskette einzulegen (diese erscheint immer dann, wenn Sie bisher noch keinen Gebrauch von der Funktion gemacht haben) erfolgt der erste Lesevorgang. Nach einiger Zeit werden Sie dann aufgefordert, die Zieldiskette einzulegen. Halten Sie dafür eine leere Diskette bereit. Diese muß nicht zuvor formatiert werden. Die Duplicate-Funktion erledigt diesen Vorgang gleich mit. Nach einigen Wechseln ist der Kopiervorgang beendet.

Für diejenigen unter Ihnen, die das Kopieren lieber vom CLI aus durchführen, sei auch dieser Weg beschrieben. Der dort zu verwendende Befehl heißt DiskCopy und wird mit zwei Parametern aufgerufen. Die Kommandozeile könnte z.B. so aussehen:

```
DiskCopy df0: to df1:
```

Damit würde vorausgesetzt werden, daß im Laufwerk DF0: die Quelldiskette und im Laufwerk DF1: die Zieldiskette liegt. Beachten Sie, daß auch dieser Befehl zuvor von der Workbench-Diskette nachgeladen werden muß! Den größten Vorteil, den das Kopieren über das CLI bietet ist der Umstand, daß man bei zwei Diskettenlaufwerken gleichzeitig liest und schreibt. Von der Workbench aus läßt sich dies nur realisieren, wenn die Zieldiskette schon einmal vorformatiert wurde.

Nach diesen Vorbereitungen, die der Datensicherheit dienten, können wir jetzt eine Arbeitsdiskette anfertigen. Da Sie natürlich schon eine Arbeitsdiskette für das GFA-BASIC besitzen, ohne das wäre der Compiler ja schließlich vollkommen sinnlos, sollte man zuerst nachschaun, ob dort nicht noch genügend Platz vorhanden ist. Sie können in diesem Fall, gleich mit dem Einrichten des Compiler-Verzeichnisses beginnen. Wir wenden uns zuerst dem Einrichten einer komplett neuen Arbeitsdiskette zu.

Die Arbeitsdiskette einrichten

Nehmen Sie sich dazu die oben erstellte Kopie der GFA-Compiler-Diskette. Dieser Diskette geben wir mit Rename (im Workbench Menü) einen neuen Namen. Wir schlagen an dieser Stelle z.B. "GFA-Workdisk" vor und werden uns immer wieder darauf beziehen. Es bleibt aber Ihnen überlassen, ob Sie auch diesen Namen wählen.

Öffnen Sie nun nach erfolgreichem Umbenennen das Hauptverzeichnis der Diskette mit einem Doppeklick. Außer dem Compiler-Symbol ist noch nichts zu sehen. Legen Sie nun die Kopie Ihrer GFA-BASIC Diskette ins Laufwerk (wir haben an dieser Stelle darauf verzichtet, zu zeigen, wie man davon eine Kopie anfertigt; einerseits sollten Sie das schon längst erledigt haben und andererseits können Sie ja oben nachschauen, der Vorgang ist der gleiche) und öffnen Sie ebenso das Hauptverzeichnis.

Nun geht es daran, den Interpreter auf die Arbeitsdiskette zu kopieren. Wir ersparen uns damit einen ständigen Wechsel zwi-

schen BASIC- und Compiler-Diskette. Dazu klicken Sie eines der Symbole an (ggf. halten Sie die <Shift>-Taste und klicken auch das zweite Symbol an) und bewegen es mit der Pfeilspitze auf das Disketten-Symbol der GFA-Workdisk. Nach dem Loslassen der linken Maustaste setzt die Workbench mit dem Kopiervorgang ein. Je nach Systemkonfiguration kann es zu bis zu vier Diskettenwechseln kommen.

Hinweis: An dieser Stelle sei angemerkt, daß es nicht sinnvoll ist, den Run-Only-Interpreter auf die Diskette zu kopieren. Schließlich werden Sie programmieren und nicht nur die Programme anwenden. Außerdem benötigt er fast nocheinmal so viel Platz, wie der komplette Interpreter. Damit wird es nachher schwierig, den Compiler vollständig einzurichten.

Nun soll - der besseren Ordnung wegen - noch eine Programm-Schublade anfertigen. Bewegen Sie dazu das Icon der Workbench-Diskette mit dem Namen "Empty" auf das Symbol der Workdisk und lassen es dort los. Nach Beendigung des "Kopiervorgangs" müssen Sie noch den Namen der neuen Schublade in "Programme" verändern. Wie das geht, haben wir bereits oben gezeigt.

Damit ist alles überstanden. Die Arbeitsdiskette ist fertig. Zur besseren Übersicht und zum Abschluß sehen Sie hier das komplette Verzeichnis der Arbeitsdiskette. Das einzige, was noch fehlt, sind die zu compilierenden Programme, die kommen später:

```
  Programme (dir)
.info                          Disk.info
GFABasic                       GFABasic.info
GfaLib                         GfaLibrary
GfaLibrary.Index               GFA_BCOM
Gl                             MakeDP
menux                          MENUX.GFA
menux.info                     Programme.info
readme.asc
```

```
Mounted disks:
Unit      Size    Used     Free Full Errs   Status    Name
DF2:      880K    1216      542  69%   0  Read/Write GFA-Workdisk
```

GFA und die Festplatte

Für alle glücklichen Besitzer einer Festplatte (man hat es da ja fast mit einer Modeerscheinung zu tun) sei an dieser Stelle noch kurz die Einrichtung des Compilers erwähnt.

1. Unterverzeichnis anlegen

Zuerst soll für eine bessere Ordnung ein entsprechendes Unterverzeichnis eingerichtet werden. Dies geschieht über die Workbench ganz einfach durch Kopieren des Empty-Drawers und dessen Umbenennung in z.B. "GFA". Sie sollten dies spätenstens jetzt tun, wenn Sie GFA kombiniert als BASIC- und Compiler-System betreiben.

2. BASIC ggf. hineinkopieren

Natürlich gehört in das GFA-Verzeichnis der Interpreter. Auch diesen können Sie über die Workbench kopieren. Bewegen Sie dazu das Symbol von der Kopie ihrer BASIC-Diskette auf das neue Schubladen-Symbol. Sollte es sich schon im Hauptverzeichnis der Festplatte befinden, können Sie es mit der gleichen Bewegung in das neue Verzeichnis legen. Ob Sie den RUN-Only-Interpreter dazunehmen, bleibt Ihnen überlassen.

3. Compiler komplett kopieren

Nun kommt endlich der Compiler an die Reihe. Diesen können wir allerdings nicht über die Workbench kopieren. Gehen Sie dazu also ins CLI und kopieren mit folgender Zeile die gesamte Compiler-Diskette in das GFA-Verzeichnis ihrer Festplatte:

```
Copy DFx:#? TO DHy:GFA
```

Für das x müssen Sie natürlich die Laufwerksnummer einsetzen, in der die Quelldiskette liegt, und für das y die Festplatten-Partitition.

4. Programme-Unterverzeichnis einrichten

Der letzte Schritt zur Ordnung führt über eine weiteres Unterverzeichnis, das diesmal in dem GFA-Verzeichnis eingerichtet wird. Hierbei duplizieren Sie wieder einmal das Empty-Icon und bewegen es auf die GFA-Schublade. Dort drin ändern Sie den Namen noch in "Programme" um. Fertig! Ihre Programme können Sie jetzt immer in diesem Verzeichnis ablegen. Für eine weitere Ordnung sei es Ihnen freigestellt, ob Sie noch tiefere Ebenen anlegen, um z.B. Spiel-, Rechen- und Zeichenprogramme zu trennen.

17.1 Beispiele und Ergebnisse

Endlich! Der Compiler ist eingerichtet, es kann losgehen. Als erstes wollen wir ein Beispielprogramm compilieren, um den Geschwindigkeitszuwachs zu beobachten.

Sie können dafür natürlich jedes beliebige (möglichst Fehlerfreie) Programm von sich verwenden. Tun Sie das ruhig. Es empfielt sich aber auch, das im folgenden abgedruckte zu verwenden, damit wir uns bei Vergleichen darauf beziehen können.

```
' ***************************************
' *                                     *
' * Programm: Apfelmännchen             *
' *                                     *
' ***************************************
'
anzfa%=4
maxfa%=anzfa%-1
xo=10              ! Zeichenbereich im
yo=5
xg=600             ! Ausgabe-Window
yg=150
x1=-2.25           ! Eckkoordinaten des
y1=-1.75
x2=1               ! Ausschnitts
y2=1.75
'
' tiefe%=4          ! Berechnugstiefe (Anzahl der Durchläufe)
' genau=2
INPUT "Bitte geben Sie die Genauigkeit ein ",genau
```

```
INPUT "Bitte geben Sie die Tiefe der Verschachtelung ein ",tiefe%
'
dx=(x2-x1)              ! Breite und Hvhe des
dy=(y2-y1)              ! Ausschnitts
'
FOR y=0 TO yg-1 STEP genau
  b=y1+dy*y/yg
  FOR x=0 TO xg-1 STEP genau*2
    a=x1+dx*x/xg
    wert=0
    c1=0.1
    c2=0.5
    flag%=1
    z%=0
    WHILE wert<anzfa% AND flag%=1
      crw1=c1
      crw2=c2
      c1=crw1*crw1-crw2*crw2+a
      c2=2*crw1*crw2+b
      wert=c1*c1+c2*c2
      INC z%
      IF z%=tiefe% THEN
        flag%=0
      ENDIF
    WEND
    IF z%=tiefe% THEN
      COLOR 0
      DRAW x,y
    ELSE
      COLOR z%-INT(z%/3)*3+1
      DRAW x,y
    ENDIF
  NEXT x
NEXT y
```

Listing 17.1: Erster Compiler Test

Nehmen Sie nun dieses oder ein anderes Programm und halten Sie es auf der Diskette zur Compilierung bereit. Wir werden nun gleich den einfachsten Weg beschreiten, um ein Programm zu compilieren. Gehen Sie dazu ins CLI oder die Shell und starten Sie auf Ihrer Diskette das Programm "menux". Dieses Programm hilft beim Erstellen von Compilaten und wird im allgemeinen als Shell bezeichnet.

Sie finden nun vor sich einen Bildschirm der wie folgt aussieht:

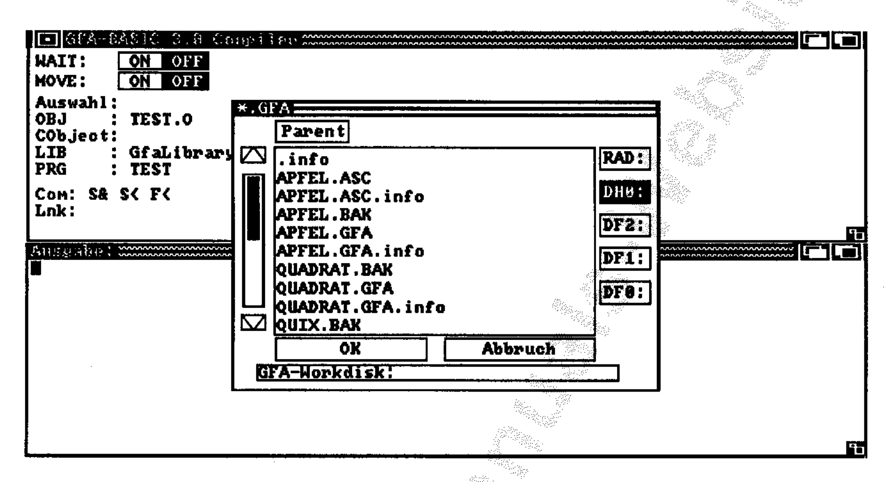


Abbildung 17.2: Die GFA-Compiler-Shell

Diese File-Select-Box dürfte Ihnen noch vom GFA-BASIC her bekannt sein. Mit ihr wird nun das Programm ausgesucht, daß compiliert werden soll. Achten Sie darauf, daß es sich um ein Programm und nicht ein ASCII-File (mit SAVE,A; Endung ".LST") handelt. Nach der Bestätigung mit OK beginnt der Compiler seine Arbeit.

Nun erkennt man auch den Zweck der Aufteilung in zwei Fenster. Das obere beinhaltet die zur Verfügung stehenden Daten, das untere gibt die Meldungen während des Compilier-Vorgangs aus. Diese könnten etwa wie folgt lauten:

```
Compiling ...
GFA-BASIC 3.0 Compiler
Zeit 2.34 Sekunden
Return: -1
Linking ...
GFA-BASIC 3.0 Linker
Zeit 7.1 Sekunden
Return: -1
```

Damit ist der gesamte Vorgang erfolgreich abgeschlossen und auf der GFA-Workdisk befindet sich unter dem Namen TEST

das compilierte Programm. Dieses kann man nun über das CLI oder auch mit Hilfe der Shell starten, um das Ergebnis zu betrachten.

Wählen Sie dazu in der Shell über das Menü "File" den Punkt "Test" oder betätigen Sie die Tasten <Control>+"T". Das File "Test" wird automatisch ausgewählt. Die Shell startet das Programm und der Ausgabebildschirm erscheint.

Es wird sich schon bei diesem kleinen Test gezeigt haben, welche Geschwindigkeitssteigerung der GFA-Compiler bietet. Allerdings bietet der Compiler selbst auf dem Weg zu dieser Steigerung noch einiges mehr, mit dem man ganz gezielt ein Programm beschleunigen kann. Es soll deshalb als nächtes die Bedienung besprochen werden.

Vorher möchten wir noch ganz deutlich darauf hinweisen, daß nach dem Compilieren das BASIC-Listing natürlich immernoch benötigt wird. Kommen Sie nicht auf die Idee, dieses vielleicht zu löschen. Der Compiler arbeitet wie eine Kodiermaschine. Nach dem Vorgang ist nichts mehr zu entschlüsseln. Eine Verbesserung oder Veränderung ist danach nicht mehr möglich.

17.2 Die Bedienung im Detail

Nicht nur zum Komfort ist eine gute Bedienung notwendig. Auch damit das Programm optimal vom Compiler übersetzt werden kann, sprich für eine effektive Ausnutzung, ist das sorgfältige Studium der Bedienungsmöglichkeiten unbedingt zu empfehlen

Der GFA-Compiler bietet neben dem Standard - ein Programm funktionsfähig zu compilieren - auch zusätzliche Optionen an, mit denen der aus einem Programm entstehende Code beeinflußt werden kann. Wir wollen uns diese Optionen ein nach der anderen ansehen. Damit diese Einstellungen möglichst leicht zu bedienen sind, werden sie über eine Shell angesteuert, die den mühsamen Weg im CLI spührbar erleichtert.

17.2.1 Auf der Workbench

Für die nun folgenden Beschreibungen starten Sie bitte aus dem CLI heraus die Shell mit der Zeile:

```
GFA-Workdisk:menux
```

Hinweis: Sicherlich ist der Weg über das CLI wesentlich weniger komfortable, als er es über die Workbench wäre. Gerade weil es ein Icon für MENUX gibt, werden Sie sich besonders über diese Vorgehensweise wundern. Unsere Arbeit mit der Shell hat aber gezeigt, daß es in einer unveränderten Version nur zu Problemen kommt, wenn man sie von der Workbench aus startet. Eine Korrektur des Programms, das dann beliebig gestartet werden kann, finden Sie unter den Programmiervorschlägen zur Shell am Ende dieses Kapitels!

Es zeigt sich der oben schon abgedruckte Bildschirm. Verlassen Sie aber entgegen dem zuerst beschriebenen Weg die File-Select-Box durch Klick auf das Feld "Abbruch" (In einer neuen Version erscheint diese zu Anfang nicht, dann entfällt dieser Punkt). Nun befinden wir uns in der Compiler-Umgebung. Im Gegensatz zur Standard-Ansteuerung im CLI bietet diese Umgebung (fast) alle Funktionen über Menüs an. Das macht die Bedienung so einfach.

Das erste, hier nicht abgebildete, Menü ist nur zum Aufruf der Programm-Informationen vorgesehen. Es gibt eine Meldung über die Version des Compilers im Ausgabefenster aus.

Hinweis: Bei der uns vorliegenden Version der Shell war ein kleiner Fehler vorhanden. Die Ausgabe der Versionsnummer erfolgte nicht im Ausgabe-Fenster, sondern im Status-Window. Das macht aber im Prinzip nichts. Sehen Sie dazu bei Gelegenheit einmal im Kapitel über die Veränderung der Shell nach.

Kommen wir jetzt zu den eigentlich wichtigen Menüs. Als erstes zeigt sich dort das File-Menü. In ihm werden alle Funktionen angeboten, die einen File-Zugriff erforderlich machen. Dies sind im Einzelnen:

1. Auswahl (<CTRL>-A)

Man wählt das Programm aus, daß compiliert werden soll. Dazu erscheint die schon bekannte File-Select-Box. Dieser Punkt wird automatisch (je nach Version der Shell) nach dem Start der Shell aufgerufen. Ebenfalls aufgerufen wird diese Funktion, wenn man den Compilier-Befehl gibt, obwohl noch kein Programm festgelegt wurde.

2. Compiler (<CTRL>-C)

Haben Sie zuerst die File-Select-Box mit "Abbruch" beantwortet, weil Sie die Compiler-Optionen verändern wollten, dann kann der Aufruf des Compilers später mit der als zweites im Menü angebotenene Funktion erfolgen.

3. Linker (<CTRL>-L)

Der Linker bindet das vom Compiler erzeugte Modul zu einem lauffähigen Programm zusammen. Diese Funktion muß unbedingt nach dem Compilieren aufgerufen werden, damit das fertige Programm erstellt werden kann. Erst nach dem Linken "versteht" der Amiga die Übersetzung als ein Programm.

4. Interpreter (<CTRL>-I)

Von der Shell aus läßt sich jederzeit mit dieser Funktion der GFA-BASIC-Interpreter aufrufen. Dadurch können Sie (genügend Speicher vorausgesetzt) kleine Änderungen an dem Programm vornehmen und es dann ein weiteres Mal compilieren. Es entfällt das lästige Verlassen der Shell, Starten des Interpreters, Verlassen des Interpreters und erneutes Starten der Shell. Allerdings kann der Amiga im Multitasking natürlich sowieso beide Programme gleichzeitig laufen lassen.

5. Test (<CTRL>-T)

Hier haben wir einen sehr praktischen Menüpunkt vor uns. Die Funktion Test führt das unter gleichem Namen vorhandene Programm sofort aus. Damit kann man im allgemeinen Fall sofort das fertige Programm anschauen und seine Funktionsweise überprüfen. Wählt man die Option Programmname=GFA-Name, sie ist bei neueren Versionen der Shell Standard-Einstellung, dann wird auch das Compilat automatisch gestartet. Man erspart sich die Auswahl des Programms.

6. Execute (<CTRL>-X)

Diese weitere Funktion arbeitet im Prinzip genauso wie die vorhergehende, jedoch kann man hier den Namen des auszuführenden Programms bestimmen. Dadurch lassen sich einerseits Programme starten, die früher schon einmal compiliert wurde. Dies kann zum Vergleich dienen, oder aber man startet irgendein anderes Tool, Utility oder eine Erweiterung, wofür keine Grenzen gesetzt sind.

Zu beachten bleibt aber, daß die GFA-Compiler-Shell so lange funktionsunfähig bleibt, wie das andere Programm läuft, da es kein Multitasking unterstützt.

7. Quit (<CTRL>-Q)

Hiermit wird der Compiler verlassen. Es findet an dieser Stelle keine Sicherheitsabfrage statt. Diese ist auch nicht nötig, da alle zu erhaltenden Daten auf dem Datenträger (Diskette, RAM-Disk oder Festplatte) gespeichert sind. Denken Sie nur daran, ggf. das Programm TEST mit Rename umzubenennen, damit es nicht bei einem neuen Compiliervorgang überschrieben wird.

Nachdem Ihnen diese Befehle im Einzelnen wohl keine Schwierigkeiten gemacht haben werden. Wollen wir uns jetzt an deren praktische Anwendung machen. Schließlich reicht es nicht zu wissen, welche Möglichkeiten es gibt, sondern man muß gleichzeitig auch erfahren, wie man diese richtig anwendet.

Als ersten und einfachsten Weg gibt es folgenden:

1. Menux übers CLI starten.
2. Programm-File (Endung .GFA) auswählen (F10).
3. Programm wird automatisch compiliert und gelinkt.
4. Programm mit Test (<CTRL>-T) starten.

Wie Sie sehen, kann man schon mit vier Schritten, die ersten drei dauern bei einem kleinen Programm nicht länger als eine Minute, wirkungsvolle Ergebnisse erzielen. Das Ergebnis im Detail hängt aber immernoch von Ihrem Programm ab.

A propos Programm. Nicht immer geht alles so reibungslos ab, wie in unserem Beispiel. Das nächste Beispiel soll zeigen, welche Probleme beim Compilieren auftreten können.

1. Menux übers CLI starten.
2. Programm-File (Endung .GFA) auswählen.
3. Das Programm wird compiliert: ein Fehler tritt auf.

An dieser Stelle wollen wir kurz einhaken. Eine Fehlermeldung tritt dann auf, wenn z.B. die Syntax des Programms fehlerhaft war. Die Ausgabe im unteren Fenster könnte dann etwa so aussehen:

```
Compiling ...
GFA-BASIC 3.0 Compiler

Programm enthält Strukturfehler
 -- mit Interpreter prüfen --
Zeit 1.24 Sekunden
Return: -1
```

Sie haben jetzt die Möglichkeit, mit dem Interpreter diesen Fehler zu korrigieren und dann erneut einen Compilier-Versuch zu starten. Hier die Reihenfolge der Vorgehensweise nach dem Auftreten des Fehlers:

4. Interpreter mit <CTRL>-I starten.
5. Programm mit der Funktion TEST in der Menüleiste durchchecken.

6. Fehler korrigieren.
7. Programm abspeichern und Interpreter wieder verlassen.
8. Erneut das Programm compilieren.
9. Programm linken.
10. Programm mit der Test-Funktion in der Shell (<CTRL>-T) aufrufen.

Fertig!

Somit haben Sie schon eine sehr häufig auftretende Fehlerquelle kennengelernt. Sicherlich gibt es noch einige Fehlermöglichkeiten mehr, doch dazu kommen wir in einem gesonderten Abschnitt.

Kommen wir nun zum zweiten Menü. Es heißt "Optionen" und wird für die Einstellungen von Compiler und Linker verwendet. Alle diese Einstellungen weichen vom Standard ab und müssen, wenn sie gewünscht werden, speziell angegeben werden. Es sind dies im einzelnen:

Interrupts

Interrupts, zu Deutsch Unterbrechungen, werden in jedem Computer(-programm) benötigt, um gleichzeitig ablaufende Operationen zu ermöglichen. Dies sind z.B. auch Tastatur-Eingaben während der Rechenphase eines beliebigen Programms. Im Normalfall nimmt das Programm selbst die Abfrage der Tastatur zu einem bestimmten Zeitpunkt in arbeit. Dies gibt aber zumeist nicht für die Tasten zum Unterbrechen eines Programms. Gleichermaßen werden die vom GFA-BASIC unterstützen periodisch auftretenden Routinen mit EVERY xx oder AFTER xx gehandhabt. Auch sie laufen auf der Basis von Unterbrechungen.

In einem Multitasking-System kann man sich dies als parallel laufende Prozesse vorstellen. Allerdings verbaucht jeder zusätzliche Prozeß neue Systemzeit, die nur in begrenztem Maße vorhanden ist. Deshalb sollte man sich beim Compilieren genau überlegen, ob es sinnvoll ist, eine Abfrage der Tastenunterbre-

chung zuzulassen. Genauso sollte die Unterbrechung nicht erfolgen, wenn in dem Programm kein EVERY xx oder AFTER xx verwendet wurde.

Zu diesem Themengebiet gehört auch eine Option, die innerhalb des Programm-Codes gesetzt wird. Man bezeichnet solche nur Teile des Programms betreffenden Optionen als Flags, die an bestimmten Stellen gesetzt und gelöscht werden können.

$U
$U+
$U-
I-
I+

Select

Die CASE-Anweisung ist eine besonders strukturierte Befehlsvariante. Mit ihr lassen sich aufwendige Fallunterscheidungen leicht bearbeiten. Allerdings vermeidet es ihre Flexibilität, daß der Compiler einen besonders schnellen Code daraus produzieren kann. Deshalb gibt es die Möglichkeit, über insgesamt zwei Optionsangaben die Compilierung zu beeinflussen.

Grundsätzlich wird die CASE-Anweisung auf Geschwindigkeit optimiert. Mit der Angabe S< läßt sich ein anderer Algorithmus bestimmen, der eine Optimierung bezüglich der Programmlänge durchführt.

Der zweite Parameter heißt S& und ändert den Umstand, daß der Compiler immer von 4-Byte-Werten ausgeht. Danach wird der Bereich auf 2-Byte-Werte eingeschränkt und kann so viel schneller bearbeitet werden.

S<
S>
S%
S&

Functions

Diese Menü-Funktion ist zusammen mit einer Flagge kombiniert, die die Kontrolle des Compilers steuert. Im Normalfall sollte eine Funktion, bevor ENDFUNC erreicht wird, mit RETURN einen Wert an den aufrufenden Programmteil zurückgeben. Dieses prüft der Interpreter in jedem Fall. Der Compiler kann sich zwischen zwei Varianten entscheiden.

Zum ersten wird die Fehlermeldung Nr. 69 ausgegeben. Dies warnt vor unbekannten Programmierfehlern. Gibt man aber die Option F< an, wird der Fehler übersprungen und die Ausführung des Programms setzt hinter der Funktion fort. Dies kann einerseits erwünscht sein, wenn er z.B. in eine weitere Funktion mündet, führt aber in den meisten Fällen zu unvorhersehbaren Aktionen.

F%
F>
F<

Procedures

Eine weitere Funktion zum Beschleunigen des Compilats! Prozeduren werden bei der Übersetzung des GFA-Compilers wiederum als Prozeduren angesehen. Dies hat die Eigenschaft, daß man sie rekursiv (aus sich selbst) aufrufen kann und die GFA-Umgebung für alle dafür notwendigen Vorkehrungen Sorge trägt. Allerdings, diese Maßnahmen erfordern einigen Rechenaufwand, der sich in der Zeit niederschlägt. Benötigt man diese Verwaltung nicht, weil weder die Rekursion noch die Verwendung von RESUME nötig ist, kann eine schnellere Art der Übersetzung gewählt werden. Dann überträgt der Compiler eine Prozedur in eine einfache 68000er-Unterroutine.

P>
P<

IntDiv

Im Normalfall gibt es keine besonderen Rechenroutinen. Bei der Verknüpfung von Integer-Zahlen eignen sich aber ganz besonders spezielle Routinen dafür, Geschwindigkeit zu gewinnen.

Mit der hierunter laufenden Einstellung ist es möglich, die Division von Integer-Variablen gesondert durchzuführen.

%0
%3

IntMul

Hier gilt das gleiche, wie auch bei der Integer-Division.

*%
*&

Error

Die Fehlermeldungen werden, sollten sie beim Compilat benötigt werden, nur als Nummern ausgegeben. Sie können dann z.B. in diesem Buch nachschlagen. Möchten Sie die erklärenden Texte haben, hilft diese Option. Das Programm wird damit aber um einiges länger, da alle Texte vorhanden sein müssen. Der Compiler "weiß" schließlich nicht, welche der Fehlermeldungen benötigt wird.

E$
E#

Memory

Für jedes Programm muß zur ordnungsgemäßen Arbeit ein sog. Stack angelegt werden. In diesem Stack werden während des Programmablaufs Daten kurzfristig abgelegt. Man kann sich das wie einen Merkzettel vorstellen, den man während einer Rechenaufgabe mit Nebenrechnungen beschreibt. Diese Zwischenspeicherung ist besonders bei vielen Prozeduren nötig. Je tiefer

also die Verschachtelung innerhalb des Programms ist, desto nötiger bedarf es eines größeren Stacks. Die Speichergröße wird mit der Memory-Angabe eingestellt. Nach dem Buchstaben "m" folgt ohne Lücke gleich die Anzahl der Bytes, die reserviert werden sollen.

mXXXX

Alle Compiler-Optionen können auch innerhalb des Programm-Codes definiert werden. Damit erspart man sich einerseits (wie Sie beim CLI später besonders merken werden) die zusätzliche Angabe vor dem Compilieren und kann außerdem eine feste Einstellung vorgeben.

DebugSym

Diese Einstellung bezieht sich nicht, wie es bei den anderen Parametern zu sehen war, auf den Compiler. Es handelt sich hierbei um einen Link-Parameter, d.h. nur der Link-Vorgang wird dadurch beeinflußt.

Will man das Programm später in seiner compilierten Form untersuchen, ist es hilfreich, wenn dazu die Bezeichnungen des Quelltextes zur Verfügung stehen. Der Linker bindet deshalb die Symbole zum Debuggen (zur Fehlersuche) in das fertige Programm ein. Sogenannte Debugger können dann während das Programm läuft anzeigen, wo man sich gerade befindet.

```
-s
```

17.2.2 Vom CLI

Die Ansteuerung des Compilers und Linkers von CLI ist eigentlich die ursprüngliche. Sie erfolgt analog zu den Aufrufen, die Ihnen aus der Shell bekannt sind. So wird der Compiler mit folgender Zeile für das Programm Test.GFA aufgerufen:

```
GFA_BCOM Test
```

Die Endung .GFA wird automatisch angehängt. Sie können sie angeben, müssen es aber nicht. Als Ausgabe erhält man bei fehlerfreiem Ablauf die schon bekannte Meldung:

```
GFA-BASIC 3.0 Compiler
```

Mehr nicht!

Möchten Sie nun eine Einstellung ändern, so schauen Sie die entsprechende Flagge nach. Sie fanden oben unter jedem Parameter eine kurze Angabe. Diese müssen Sie nun hinter dem File-Namen angeben. Sollen z.B. die Fehlermeldungen als Texte ausgegeben werden, sieht die Compiler-Zeile wie folgt aus:

```
GFA_BCOM Test E$
```

Jede weitere Flagge wird getrennt durch eine Lücke hinter den File-Namen geschrieben. Sehen Sie in der folgenden Tabelle eine Übersicht aller Flaggen und ihrer Bedeutung:

Flag	Bedeutung
U	Die Abbruch-Tastenkombination (<Control>-<SHIFT>-<ALT>) wird nur einmal überprüft.
U+	Nach jedem Befehl wird ein Tastatur-Abbruch überprüft.
U-	Die Prüfung des Tastatur-Abbruchs entfällt.
I+	Interrupt-Routinen sind eingeschaltet.
I-	Interrupt-Routinen sind ausgeschaltet.
%0	Integer-Division wird nur bei Übertragung in eine Integer-Variable durchgeführt.
%3	Integer-Division wird immer als Integer-Division durchgeführt.
%6	Integer-Addition vermischt mit anderen Zahlentypen wird als Fließkomma-Addition durchgeführt.
N+	Die zusätzliche Überlaufprüfung des Compilers wird eingeschaltet.
N-	Es findet keine zusätzliche Überlaufprüfung statt.
*&	Langwort-Multiplikationen werden mit dem speziellen Assembler-Befehl "muls" durchgeführt.
*%	Langwort-Multiplikationen werden mit der Unterroutine LMUL durchgeführt.
E$	Fehlermeldungen werden als Text ausgegeben.
E#	Fehlermeldugnen werden als Nummer ausgegeben.

Flag	Bedeutung
F%	Die von Funktionen zurückgegeben Parameter werden als Integer interpretiert.
F>	Erzeugt ENDFUNC als Funktionsabschluß.
F<	Erzeugt keine ENDFUNC bei Funktionen.
P>	Unterroutinen werden von GFA-BASIC verwaltet.
P<	Unterroutinen werden als 68000er-Unterroutinen gehandhabt.
S&	Die Parameter von SELECT werden als 2-Byte-Werte verwaltet.
S%	Die Parameter von SELECT werden als 4-Byte-Werte verwaltet.
S<	SELECT-Auswahl wird auf Programmlänge hin compiliert.
S>	SELECT-Auswahl wird auf Geschwindigkeit hin compiliert.
C+	Die Register A3-A6 werden beim Aufrufen von Assembler-Routinen mit CALL gerettet.
C-	Die Register A3-A6 werden nicht gerettet.
M xxxx	Reserviert xxxx Bytes separaten Speicher.
X name	Bindet die Routine "name" aus einer Link-Datei ein.

Bisher noch nicht dokumentiert wurden die folgenden Flaggen:

Unterroutinen einbinden

In manchen Situationen eignet es sich besser, nicht in BASIC zu programmieren und dafür eine Routine aus einer anderen Programmiersprache zu wählen. Dies mag daran liegen, daß es in BASIC zu kompliziert wäre oder aber auch daß BASIC nicht schnell genug ist.

In diesem Fall ersetzt der GFA-Linker aufgrund der Markierung im Quelltext diese Routine mit der in der GFA-Library vorhandenen. Dazu finden sie mehr Hinweise im Kapitel "Fortgeschrittene Compiler-Nutzung".

```
X name
```

Registerverwaltung

Die schon als Standard bekannte Möglichkeit, innerhalb der GFA-BASIC Programme Assembler-Routinen anzuspringen, erfährt durch diese Parameter neuen Komfort. Im Normalfall

dürfen die Register A3-A6 nicht verwendet werden, da sie wichtige Werte enthalten. Mit der Option C+ wird eine automatische Speicherung eingestellt, die zwar mehr Speicherplatz und Zeit verbraucht, dafür aber Sicherheit garantiert.

```
C+
C-
```

Überlaufprüfung

Aufgrund der Organisation des Compilers gibt es Fehlermöglichkeiten, die durch fehlende Abfragen entstehen können. Dies liegt an dem Bestreben, möglichst schnelle Compilate zu erzeugen. So wird die Abbruchbedingung der Schleife FOR-TO-NEXT (Erreichen des letzten Zahlenwertes) erst nach dem NEXT durchgeführt. Hierbei wurde die Abfrage vor dem Erhöhen der Laufvariablen eingespart. Dies kann aber besonders bei Integer-Variablen zu Problemen führen, da nach dem höhsten Wert wieder der kleinste folgt, sich somit also eine Endlosschleife ergibt. Durch die Option N+ fügt der Compiler eine zusätzliche Abfrage ein, die diese Endlosschleifen verhindert.

```
N+
N-
```

Fließkommaaddition

Leider treten bei der Verbindung von Fließkomma- und Integer-Addition Rundungsfehler auf. Dies liegt an der Konvertierung der Fließkommazahlen zu Intergerwerten, bei der die Nachkommastellen überlicherweise abgeschnitten werden. Führt dies zu starken Berechnungsfehlern, kann mit der Option %6 eine Umwandlung der Werte in Fließkommanzahlen erzwungen werden. Es ist zu beachten, daß damit ein erheblicher Geschwindigkeitsnachteil erzeugt wird. Verwenden Sie diese Angabe deshalb nur für ganz bestimmte Programmteile. Die teilweise Nutzung der Compiler-Flaggen wird im Kapitel über die fortgeschrittene Compiler-Nutzung erklärt.

```
%6
```

Damit sind Sie über alle Möglichkeiten zur Einstellung bei der Compilierung informiert. Es soll jetzt der Link-Vorgang betrachtet werden, bei dem weitere Einstellungen möglich sind.

Nach dem Compilier-Vorgang folgt zwangsläufig das Linking. Dieses wird als separates Programm aufgerufen. Das Programm bindet, wie oben schon erwähnt, das Programm-Modul (es können auch mehrere sein) zu einem lauffähigen Programm zusammen. Der Linker wird wiederum mit dem Programm-Namen bzw. dem Namen des Objekt-Moduls aufgerufen:

```
GL Test
```

Diese Befehlszeile reicht im Normalfall aus. Jedoch gibt es für den Linker genauso wie für den Compiler weitere Optionen, die Spezialfälle zulassen. Sehen Sie auch dazu eine Tabelle:

Flagge	Bedeutung
-s	Die Symboltabelle wird dazugelinkt.
+Lib	Für das Linking wird die Library "Lib" verwendet und nicht die standard-eingestellte "GFALibrary".
-Lname	Entspricht dem obigen.
Name	Fügt zusätzlich die Objekt-Datei "Name.O" zum Standard-Objekt "TEST.O".
#	Die Objekt-Datei "TEST.O" wird nicht dazugelinkt.
-Oname	Ersetzt den Namen "TEST.O" durch "name.O" als Haupt-Linkdatei.
-Pname	Nennt das erzeugte Programme "name" und nicht "TEST".
-W	Schaltet die Wait-Flag ein, damit auf Diskettenwechsel gewartet wird.

17.2.3 Die Fehlermeldungen des Linkers

?... Das oben nach dem Fragezeichen genannte Symbol ist nicht bekannt. Diese Fehlermeldung kann auftreten, wenn im Quellprogramm über die Option

```
$ X name
```

eine externe Funktion eingebunden werden soll, diese aber beim Linking vergessen wurde.

Zur Behebung des Fehlers kann entweder untersucht werden, ob vergessen wurde, die dazugehörige Link-Datei anzugeben oder aber der Name im Quelltext falsch geschrieben wurde.

+... Das angegebene Symbol wurde zweimal definiert. Innerhalb des gesamten Objekt-Codes existiert zweimal die gleiche Funktion. Das kann z.B. daran liegen, daß innerhalb zweier dazugefügter Objekt-Dateien das gleiche Untermodul definiert wurde.

Entfernen Sie zur Fehlerkorrektur das doppelte Untermodul in einem der Objekt-Sources. Handelt es sich nicht um den gleichen Code, müssen Sie die Namen ändern, damit der Fehler nicht ein zweites Mal auftritt.

>... Ein 16-Bit-Offset ist zu groß. Der Abstand zwischen Daten und dem Programmteil, der auf diese Daten zugreift ist zu groß. Das kann neben dem Auslesen von Daten auch der Sprung in eine Routine sein, die zu weit weg liegt.

Eine Möglichkeit, die leider nicht immer hilft, ist das Linken des Programms in anderer Reihenfolge der Objekt-Module. Dabei muß auf die Beziehung der einzelnen Routinen geachtet werden. Allerdings helfen hier nur Details weiter, die ganz speziell vom Source-Code abhängen. Sorry.

17.3 Effektives Compiler-BASIC

Sicherlich sollte sich der Einsatz eines Compilers auch lohnen. Programme sollen schneller und damit leistungsfähiger werden. Grundsätzlich wird aber nicht jedes Programm meßbar schneller. Dies liegt ganz einfach daran, daß das GFA-BASIC in manchen Bereichen schon die maximale Geschwindigkeit ausgenutzt hat.

An dem folgenden Beispiel sehen Sie ein optisch sehr schönes Programm, daß leider in der Compilierung nur wenig schneller wird:

```
' Demonstrationsprogramm für Compiler
'
' Obwohl dieses Programm grundsätzlich nicht sehr schnell ist,
' kann es mit dem Compiler nur wenig beschleunigt werden.
' Dies zeigt die Zeitangabe im Window.
'
' (c) 1989 by DATA BECKER, Düsseldorf
' (p) 1989 by Wgb, Großhansdorf
'
DEFINT "a-z"
'
COLOR 1
begin%=1
ende%=360
breite%=320
hoehe%=128
'
DRAW 16000,600
WHILE INKEY$=""
  t1%=TIMER
  FOR i%=begin% TO ende% STEP 1
    DRAW  TO 320+SIN(i%)*(breite%/(i%/50)),128+COS(i%)*(hoehe%/(i%/50))
    COLOR i% AND 3
  NEXT i%
  FOR i%=ende% TO begin% STEP -1
    DRAW  TO 320+SIN(i%)*(320/(i%/50)),128+COS(i%)*(hoehe%/(i%/50))
    COLOR i% AND 3
  NEXT i%
  t2%=TIMER
  PRINT AT(0,0);(t2%-t1%)/200
WEND
```

Listing 17.2: Stern-Zeichner

Wie Sie sehen, besteht ein großteil der Arbeit, die dieses Programm verrichtet, darin, zu zeichnen. Gerade aber diese Arbeiten werden vom Betriebssystem durchgeführt. Sie können nicht beschleunigt werden. Diese Gesetzmäßigkeit läßt sich auf alle Ein- und Ausgaben übertragen. So werden Grafiken nicht schneller gezeichnet, wenn man das Programm compiliert.

Was sich allerdings steigern läßt sind aufwendige Berechnungen. Hier werden sehr auffällige Geschwindigkeitssteigerungen registriert. Nehmen wir dazu zum Beispiel die Primzahlenberechnung nach dem Verfahren "Das Sieb des Erathostenes". Dabei wird ein Feld mit allen zu untersuchenden Zahlen definiert.

Dieses Feld geht man nun Zahl für Zahl durch und entfernt alle Vielfachen der Zahl. Nun geht man zur nächsten vorhandenen Zahl und fährt so fort. Zum Schluß bleiben nur noch ganz wenige der Zahlen. Das Verfahren kann bei der Hälfte der Menge abgebrochen werden, da alle übrigen Zahlen nicht mehr als Vielfaches vorhanden sind.

Das nun abgedruckte Listing verwendet dieses Verfahrung und wurde unter Ausnutzung der GFA-BASIC Eigenheiten programmiert. Dadurch ist es schon in BASIC besonders schnell. Aber warten Sie ab ...

```
' Das Sieb des Erathostenes
'
' Programm zur Demonstration der
' Beschleunigung der Rechengeschwindigkeit
'
' (c) 1989 by DATA BECKER, Düsseldorf
' (p) 1989 by Wgb, Großhansdorf
'
DEFINT "a-z"
anzahl%=2000
abbruch%=anzahl%/2
DIM feld%(anzahl%)
zaehler%=2
t1%=TIMER
'
WHILE zaehler%<abbruch%
  '  PRINT zaehler%
  anz%=2
  WHILE zaehler%*anz%<=anzahl%
    entfernen%=zaehler%*anz%
    feld%(entfernen%)=1
    INC anz%
  WEND
  INC zaehler%
  WHILE feld%(zaehler%) AND zaehler%<=abbruch%
    INC zaehler%
  WEND
WEND
'
t2%=TIMER
PRINT "Es wurden ";(t2%-t1%)/200;" Sekunden für ";anzahl%;" Primzahlen
benvtigt"
WHILE INKEY$=""
WEND
FOR i%=1 TO anzahl%
  IF feld%(i%)=0 THEN
```

```
    PRINT i%
  ENDIF
NEXT i%
```

Listing 17.3: Das Sieb des Erathostenes

Dieses doch recht einfache Programm vermeidet innerhalb der Schleifen jede Standard-Berechnung. Aus diesem Grund wird die Abbruch-Bedingung <anzahl%/2 vorher berechnet und in abbruch% niedergeschrieben.

Compilieren Sie nun dieses Programm mit den Standard-Optionen. Sie werden einen enormen Geschwindigkeitsgewinn feststellen. Bei einem Feld von 2000 Zahlen benötigt der Interpreter 3.8 Sekunden. In der compilieren Version werden maximal 0.4 Sekunden gebraucht! Das ist das 9,5 Fache!

Es lassen sich aber noch weitere Steigerungen erzielen. Dazu kann z.B. die Multiplikation, die sich alleine auf Integer-Zahlen beschränkt, mit einer besonders schnellen Routine durchgeführt werden. Setzen Sie dazu im Menü "Optionen" das Flag "IntMul". Die Verbesserung macht sich deutlich bemerkbar.

Ganz allgemein läßt sich sagen, daß es besonders sinnvoll ist, die Verfahren, mit denen gerechnet wird, gut auszuwählen. Dies hängt aber in erster Linie von den Variablen ab. Man sollte sich deshalb angewöhnen, für ganze Zahlen keine Fließkomma-Variablen, sondern Integer-Variablen zu verwenden. Dies macht es für den Compiler einfacher, eine Geschwindigkeitssteigerung durchzuführen. Aber auch hier sollten Sie noch genau auf die Bereiche achten. Benötigen Sie wirklich 4-Byte-Integer oder reichen 2-, wenn nicht gar 1-Byte aus.

17.4 Fortgeschrittene Compiler-Nutzung

Compiler-Optionen innerhalb des Quelltextes

Wie Sie bei der Beschreibung der Compiler-Shell gesehen haben, führen verschiedene Flaggen zu anderen Ergebnissen im Compi-

lat. Diese Flaggen haben aber nicht alle globalen Charakter. Das heißt es können auch nur Teile eines Programms durch diese Einstellungen beeinflußt werden. Anhand einiger Beispiele soll dies jetzt gezeigt werden.

Grundsätzlich ist es auch möglich, Optionen innerhalb des Programms zu wählen. Sie können also die Einstellungen, die sie mit der Shell über die Menüs anwählen, genausogut in den Programmtext eintragen. Will man z.B. das Programm durch die Tastenkombination <CTRL>-<SHIFT>-<ALT> unterbrechbar machen, setzt man an den Anfang des Listings die Zeile:

```
$ U+
```

Damit wird die Unterbrechung zugelassen. Das ist besonders sinnvoll, wenn es sich um eine Endlosschleife im Hauptprogramm handelt, die unterbrochen werden muß. Aber auch bei vermuteten Fehlern, die zu Endlosdurchläufen führen, ist diese Option hilfreich.

Optionen für Programmabschnitte

Das nun folgende Programm macht an zwei Stellen das gleiche. Es addiert zu einer Variablen immer wieder Zufallszahlen im Bereich von 0-1,9. Da es sich um eine ganzzahlige Variable handelt, wird die Wahrscheinlich in gleichen Teilen zu 0 und 1 stehen, da immer abgerundet wird. Allerdings ist die Ausführungsgeschwindigkeit sehr unterschiedlich.

```
' Compiler-Test
'
' Optionen innerhalb des Quell-Textes
'
' (c) 1989 by DATA BECKER, Düsseldorf
' (p) 1989 by Wgb, Großhansdorf
'
DEFINT "a-z"
PAUSE 10
'
$ %6
t1%=TIMER
CLR a%
FOR i%=1 TO 10000
  a%=a%+(RND(1)*2)
```

```
NEXT i%
t2%=TIMER
'
$ %3
CLR a%
FOR j%=1 TO 10000
  a5%=a%+(RND(1)*2)
NEXT j%
t3%=TIMER
'
PRINT "1. Schleife: ";(t2%-t1%)
PRINT "2. Schleife: ";(t3%-t2%)
WHILE INKEY$=""
WEND
```

Wenn Sie dieses Programm compilieren werden Sie unabhängig von den Einstellungen über die Integer-Addition ein Ergebnis erhalten, daß die erste Schleife wesentlich langsamer klassifiziert. Dies ist auch verständlich, denn die dortige Addition wird in Fließkommanzahlen durchgeführt und erst danach in das Integer-Format umgewandelt. In manchen Fällen ist es aber nötig, diese Addition zu wählen. Dann können Sie den entsprechenden Bereich über die Flaggen kennzeichnen.

17.4.1 Ergänzungen für die Compiler-Shell

Nachfolgend sehen Sie das Listing der Compiler-Shell, die im Mittelpunkt dieses Kapitels steht. Auf der uns vorliegenden Version befanden sich einige Fehler, die wir anhand des abgedruckten Quelltextes korrigieren wollen. Außerdem haben Sie die Möglichkeit, einige Verbesserungen einzubauen.

```
$ m2000
$ s&,s<,f<
.pl96
.n9
'
RESERVE 2000
' Programmnamen
gfaint$="GFABasic"
gfacom$="GFA_BCOM"
gfalnk$="GL"
'
' Environment Variablen
tobj$="TEST.O"              !Erzeugtes O-File
```

```
tprg$="TEST"                 !Erzeugtes PRG
tlib$="GfaLibrary"             !Name der Library
cobj$=""                     !zusätzliche O-Files für Linker
'
' Aktueller Pfad
p$=DIR$(0)
'
DIM a$(50)
FOR i%=0 TO 50
  READ a$(i%)
  EXIT IF a$(i%)="*"
NEXT i%
a$(i%)=""
'
coi&=0          !kein I
cos&=3          !S& und s<
cof&=1          !F<
cod&=0          !kein %3
com&=0          !kein *&
coe&=0          !kein E
cop&=0          !kein P>
dbsym&=0        !keine DebugSymbole
auto&=1         !xxx aus xxx.gfa
'
DATA "About "
DATA  GFA-BASIC 3.0 Compiler
DATA
DATA "File "
DATA  Auswahl     ^A
DATA  Compiler    ^C
DATA  Linker      ^L
DATA  Interpreter ^I
DATA ----------------
DATA  Test        ^T
DATA  Execute     ^X
DATA ----------------
DATA  Quit        ^Q
DATA
DATA "Optionen "
DATA +I Interrupts
DATA +S Select
DATA +F Functions
DATA +P Procedures
DATA +/ IntDiv
DATA +* IntMul
DATA +E Error
DATA ----------------
DATA +M Memory
DATA ----------------
DATA +D DebugSym
DATA
DATA Sets
DATA   G3WAIT   W
```

```
DATA   G3MOVE   M
DATA -------------
DATA   G3OBJ    O
DATA   G3PRG    P
DATA   G3LIB    L
DATA   PRG=GFA  F2
DATA   C-Object C
DATA *
'
maxy&=255
s$="GFA-BASIC 3.0 Compiler"
sss$="CON:0/"+STR$(INT(maxy&/2))+"/640/"+STR$(INT(maxy&/2))+"/Ausgabe:"
OPEN "o",#1,sss$
OPENW #0,0,0,639,maxy&/2-1,0,&H100F
TITLEW #0,s$,s$
MENU a$()
@menu34
ON MENU GOSUB men
com_opt
'
DO
  ON MENU
  i$=INKEY$
  IF i$<>""
    a%=BYTE{V:i$}
    key
  ENDIF
  IF MOUSEK=1 AND MOUSEX<138 AND MOUSEX>70
    IF MOUSEX<75+28
      fl!=TRUE
    ELSE
      fl!=FALSE
    ENDIF
    IF MOUSEY>12
      IF MOUSEY<24
        twait!=fl!
        com_opt
      ELSE IF MOUSEY<34
        tmove!=fl!
        com_opt
      ENDIF
    ENDIF
    WHILE MOUSEK
    WEND
  ENDIF
LOOP
'
> PROCEDURE men
  check(-MENU(0))
RETURN
> PROCEDURE key
  SELECT BYTE(a%) AND &HDF       !diese ascii-codes ohne scan-code
  CASE "*","/","O","L","P","M","W","C"
```

```
    a%=BYTE(a%) AND &HDF
    check(a%)
  CASE 127    !delete (amiga has no undo-key !!!)
    check(127)
  CASE 155
    a|=BYTE{SUCC(V:i$)}
    SELECT a|
    CASE 63 !Help
      check(1000)
    DEFAULT
      check(ADD(1000-47,a|))
    ENDSELECT
  CASE 1 TO 26  !Control+Key
    check(a%)
  ENDSELECT
RETURN
> PROCEDURE check(x%)
  SELECT x%
  CASE -1
    PRINT #1,"Version 3.0"
    RELSEEK #1,0
  CASE -4,1001,1        !^A
    do_fsel(".GFA",f$)
    IF auto& AND 1
      IF RIGHT$(f$,4)=".GFA"
        z$=LEFT$(f$,LEN(f$)-4)
        IF z$<>tprg$
          tprg$=z$
          '         com_opt
        ENDIF
      ENDIF
    ENDIF
  CASE -5,3             !^C
    IF @chk_file(f$)
      keyclr
      compile
    ENDIF
  CASE 1010 !F10
    IF @chk_file(f$)
      compile
      link
    ENDIF
  CASE -6,12  !^L
    link
  CASE -7,9   !^I
    IF LEN(f$)
      f1$=" - "+f$+" "
    ELSE
      f1$=" "
    ENDIF
    PRINT #1,"Starting ";gfaint$;f1$
    RELSEEK #1,0        !clear buffer
    t%=TIMER
```

```
    EXEC gfaint$+f1$,0,0
    tmx
  CASE -9,20 !^T
    PRINT #1,"Executing ";tprg$
    RELSEEK #1,0        !clear buffer
    t%=TIMER
    e%=EXEC(tprg$,0,0)
    tmx
  CASE -10,24 !^X
    do_fsel("Programm:",x$)
    IF LEN(x$)
      PRINT #1,"Executing  ";x$
      RELSEEK #1,0        !clear buffer
      t%=TIMER
      e%=EXEC(x$,0,0)
      tmx
    ENDIF
  CASE -12,17 !^Q
    END
  CASE -15  !I
    INC coi&
  CASE -16  !S
    INC cos&
  CASE -17  !F
    INC cof&
  CASE -18  !P
    INC cop&
  CASE -19  !/
    INC cod&
  CASE -20  !*
    INC com&
  CASE -21  !E
    INC coe&
  CASE -23  !M
    in(m$)
    n=INT(VAL(m$))
    IF n>1000000000 OR n<=0
      m$=""
    ELSE
      m$=STR$(n)
    ENDIF
  CASE -25      !D
    INC dbsym&
  CASE -28,"W"
    twait!=NOT (twait!)
    com_opt
  CASE -29,"M"
    tmove!=NOT (tmove!)
    com_opt
  CASE -31,"O"
    in(tobj$)
  CASE -32,"P"
    in(tprg$)
```

```
  CASE -33,"L"
    in(tlib$)
  CASE 127,1000 !undo,help
    com_opt
  CASE -34,1002 !F2
    INC auto&
  CASE -35,"C"
    in(cobj$)
  ENDSELECT
  MENU a$()
  menu34
  ON MENU GOSUB men
  ON MENU KEY GOSUB key
  com_opt
  keyclr
RETURN
> PROCEDURE com_opt
  co$=""
  IF com& AND 1
    co$=co$+" *&"
  ENDIF
  IF cod& AND 1
    co$=co$+" %3"
  ENDIF
  IF cos& AND 1
    co$=co$+" S&"
  ENDIF
  IF cos& AND 2
    co$=co$+" S<"
  ENDIF
  IF coe& AND 1
    co$=co$+" E$"
  ENDIF
  IF coe& AND 2
    co$=co$+" E-"
  ENDIF
  IF coi& AND 1
    co$=co$+" I+"
  ENDIF
  IF cof& AND 1
    co$=co$+" F<"
  ENDIF
  IF cop& AND 1
    co$=co$+" P>"
  ENDIF
  IF LEN(m$)
    co$=co$+" m"+m$
  ENDIF
  IF auto& AND 1
    IF RIGHT$(f$,4)=".GFA"
      z$=LEFT$(f$,LEN(f$)-4)
      IF z$<>tprg$
        tprg$=z$
```

```
      '        com_opt
      ENDIF
    ENDIF
  ENDIF
  '
  COLOR 0
  PBOX 70,12,138,34
  COLOR 1
  TEXT 7,100,"Com:"+co$+SPACE$(10)
  TEXT 7,110,"Lnk:     "
  IF dbsym& AND 1
    TEXT 43,110,"-s"
  ENDIF
  '
  TEXT 7,20,"WAIT:    ON  OFF"
  TEXT 7,32,"MOVE:    ON  OFF"
  BOX 70,12,138,22
  BOX 70,24,138,34
  invert(12,twait!)
  invert(24,tmove!)
  sss$=SPACE$(8)
  LINE 75+28,12,75+28,22
  LINE 75+28,24,75+28,34
  TEXT 7,45,"Auswahl: "+f$+sss$
  TEXT 7,55,"OBJ     : "+tobj$+sss$
  TEXT 7,65,"CObject: "+cobj$+sss$
  TEXT 7,75,"LIB     : "+tlib$+sss$
  TEXT 7,85,"PRG     : "+tprg$+sss$
RETURN
> PROCEDURE invert(y0&,f!)
  LOCAL y1&,a$,x0&,x1&
  INC y0&
  y1&=y0&+8
  IF f!
    x0&=71
    x1&=75+27
  ELSE
    x0&=75+29
    x1&=137
  ENDIF
  GET x0&,y0&,x1&,y1&,a$
  PUT x0&,y0&,a$,&H30
RETURN
> PROCEDURE do_fsel(x$,VAR f$)
  CLR f$
  FILESELECT x$,"OK",p$,f$
  f$=TRIM$(f$)
  com_opt
RETURN
> PROCEDURE tmx
  PRINT #1,"Zeit ";(TIMER-t%)/200;" Sekunden"
  PRINT #1,"Return: ";e%
  RELSEEK #1,0        !clear buffer
```

```
RETURN
> PROCEDURE link
  PRINT #1,"Linking   ... "
  RELSEEK #1,0         !clear buffer
  t%=TIMER
  IF dbsym& AND 1
    s$=" -s "
  ELSE
    s$=" "
  ENDIF
  env
  sss$=s$+cobj$+env$+CHR$(10)
  e%=EXEC(gfalnk$+sss$,0,1)
  tmx
  keyclr
RETURN
> PROCEDURE compile
  PRINT #1,"Compiling ... "
  RELSEEK #1,0         !clear buffer
  t%=TIMER
  env
  sss$=" "+f$+co$+env$+CHR$(10)
  i_on
  e%=EXEC(gfacom$+sss$,0,1)
  i_off
  tmx
RETURN
> PROCEDURE env
  env$=" -O "+tobj$+" -P "+tprg$+" +"+tlib$
  IF twait!
    env$=env$+" -W"
  ENDIF
  IF tmove!
    env$=env$+" -M"
  ENDIF
RETURN
> PROCEDURE in(VAR a$)
  OPENW #1,70,105,500,25,0,15+4096
  TITLEW #1,"Eingabe:"
  FORM INPUT 60 AS a$
  CLOSEW #1
  a$=TRIM$(a$)
RETURN
> PROCEDURE i_on
RETURN
> PROCEDURE i_off
RETURN
> PROCEDURE keyclr
  WHILE INKEY$<>""
  WEND
RETURN
> PROCEDURE menu34
  IF auto& AND 1
```

```
    menu34&=&H52+&H101
  ELSE
    menu34&=&H52
  ENDIF
  MENU 34,menu34&
RETURN
'
FUNCTION chk_file(VAR f$)
  $ F%
  IF LEN(f$)
    IF EXIST(f$)
      RETURN TRUE
    ENDIF
  ENDIF
  do_fsel(".GFA",f$)
  IF LEN(f$)
    IF EXIST(f$)
      RETURN TRUE
    ENDIF
  ENDIF
  RETURN FALSE
ENDFUNC
```

Listing 17.4: Die Compiler-Shell "MenuX"

Wie Sie an den Procedure-Pfeilen sehen, unterteilt sich das Listing in folgende Gruppen:

```
> PROCEDURE men
> PROCEDURE key
> PROCEDURE check(x%)
> PROCEDURE com_opt
> PROCEDURE invert(y0&,f!)
> PROCEDURE do_fsel(x$,VAR f$)
> PROCEDURE tmx
> PROCEDURE link
> PROCEDURE compile
> PROCEDURE env
> PROCEDURE in(VAR a$)
> PROCEDURE i_on
> PROCEDURE i_off
> PROCEDURE keyclr
> PROCEDURE menu34
```

Jede einzelne Procedure erledigt eine Aufgabe, die sich bei den meisten schon am Namen ablesen läßt. Wir wollen diese Aufgabe je nach Wichtigkeit nun besprechen.

Das Hauptprogramm-Modul dient zur Initialisierung der Stammdaten und des Menü-Systems. Alle Ergänzungen, die die Funktionen der Shell betreffen, müssen in diesen Menü-Daten verankert werden. Dazu gibt es später einige Beispiele. Widmen wir uns zuerst der Einstellung eigener Standard-Parameter. Die Festlegung finden wir in den folgenden Zeilen:

```
coi&=0          !kein I
cos&=3          !S& und s<
cof&=1          !F<
cod&=0          !kein %3
com&=0          !kein *&
coe&=0          !kein E
cop&=0          !kein P>
dbsym&=0        !keine DebugSymbole
auto&=1         !xxx aus xxx.gfa
'
```

Programm-Ausschnitt 17.5: Die Environment Variablen

Die erste Variable coi& steht für Interrupts. Setzen Sie hier eine 1 hin, wenn Interrupts (s.o.) zugelassen werden sollen.

Cos& steht für die Select-Optimierung. Eine 1 bedeutet eine Verwaltung der Select-Parameter als 2-Byte-Variable. Standard ist die Verwaltung als 4-Byte-Variable. Setzen Sie dort eine 2, so wird der Programmtext bezüglich der Programmlänge und nicht wie sonst bezüglich der Geschwindigkeit optimiert. Sie können beide Optionen addieren, um sie gleichzeitig zu erhalten. So ist es auch im Original eingestellt.

Cof& steht für die Funktions-Option. Eine 1 bedeutet, daß ENDFUNC nicht erzeugt wird.

Cod& steht für die Integer-Addition. Der Wert 1 würde die Flag %3 setzen, die eine Integeraddition erzwingt.

Com& steht für die Integer-Multiplikation. Auch hier bedeutet eine 1, daß *& gesetzt ist und damit Langwortmultiplikationen mit dem Assembler-Befehl muls durchgeführt werden.

Coe& ist die Error-Flagge, eine 1 steht für Fehlermeldungen als Text und die 2 für Fehlermeldungen-Unterdrückung (E-).

Cop& verwaltet die Prozeduren Flag. Setzt man eine 1 ein, werden die Unterroutinen, die sonst als 68000er Routinen laufen, als von GFA-BASIC verwaltete Unterprogramme compiliert.

dbsym& dient als Linker-Flag. Bei einer gesetzten 1 wird zum späteren Programm auch die Symbol-Tabelle gelinkt. Dies verbraucht wesentlich mehr Speicherplatz.

Die auto&-Flagge sorgt nicht dafür, daß die Shell wegfährt, sondern ist für die automatische File-Namen-Umsetzung zuständig.

Die Prozedur men ist für die Menü-Abfrage zuständig. Sie dient als Anlaufstelle für das im Hauptprogramm aufgerufene

```
ON MENUE GOSUB men
```

und beinhaltet nur eine Abfrage der Menüpunkte, die im entsprechenden Fall den negativen Wert an die Prozedur Check() weitergibt.

Die Prozedur key ist für die Tastaturabfrage. Hierbei werden alle Sondertasten ausgemustert. Einfache Tastendrücke bekommen die Zahlen ab 1000.

Die Prozedur check(x%). Hier werden die Weichen für die Funktionen des Programms gestellt. Zu jeder möglichen Taste existiert in der SELECT-Auswahl ein CASE, das die entsprechenden Aktionen auslöst.

Die Prozedur com_opt ist für die Text-Gestaltung der Options-Flags zuständig. Hier wird überprüft, ob die nötigen Variablen gesetzt wurden und der String um den nötigen Text ergänzt.

Die Prozedur invert(y0&,f!) sorgt für die Invertierung der Kästen WAIT und MOVE im Options-Window.

Die Prozedur do_fsel(x$,VAR f$) führt den Aufruf der File-Select-Box durch und liefert den ausgewählten File-String zurück.

Die Prozedur tmx bemißt die Zeit während einer Zeitschleife. Dafür wird zu Anfang immer die Variable t% auf TIMER gesetzt. Innerhalb der Prozedur wird nur die Differenz zwischen t% und dem aktuellen TIMER berechnet.

Die Prozedur link erledigt das Linking. Dafür wird in den String sss$ das komplette Argument für den Link-Befehl zusammengebaut. Dieses besteht aus der Compiler-Option, der Objekt-Liste, den Umgebungs-Einstellungen (siehe Prozedur env) und dem Textabschluß CHR$(10). Diese wird zusammen mit dem Befehlsnamen über EXEC aufgerufen.

Die Prozedur compile erledigt das Compilieren. Auch hier wird in der Varible sss$ das gesamte Argument zusammengesetzt. Es besteht hier aus dem File-Namen, den unter com_opt berechneten Optionen und den bekannten Umgebungs-Einstellungen mit dem Textabschluß.

Die Prozedur env: Diese bisher noch nicht dokumentierte Prozedur erstellt einen String, der immer die Angaben über das Objekt-File ("-O "+tobj$), das Programm-File ("-P "+tprg$) und die Compiler-Library ("+"+tlib$) enthält. Diese Angaben werden für beide, Compiler und Linker, benötigt.

Die Prozedur in öffnet ein kleines Window in der Mitte des Bildschirms, in das ein Text eingegeben werden kann.

Die Prozeduren i_on und i_off enthalten selbst keine Befehle, wahrscheinlich wurden sie nur innerhalb der Testphase gebraucht. Eine Anwendung ist zur Zeit nicht bekannt.

Die Prozedur keyclr dient, wie der Name schon sagt, dazu den Tastaturpuffer zu löschen. Dies geschieht auf einfach Weise: Es werden so lange Zeichen eingelesen, bis keine mehr vorhanden sind. Der Puffer ist leer.

Die Prozedur menu34 verwaltet den Menüpunkt mit der Nummer 34. Dies ist die Funktion zur Auswahl, ob das Programm später den gleichen Namen wie das ursprünglich ausgewählte File haben soll. Da die Menü-Verwaltung des GFA-BASIC das Abhaken nicht übernimmt, wird es von dieser Funktion übernommen.

Korrektur der Fehler

Leider bestehen innerhalb des Programms einige kleine Programmierfehler, die wir an dieser Stelle korrigieren wollen.

Die Versionsnummer wurde in der uns vorliegenden Version nicht im Ausgabe-Fenster, sondern im Status-Fenster ausgegeben. Dies ist nicht nur unschön, sondern enthält uns auch eine Information vor.

Deshalb kann man in der Prozedur mit zwei kleinen Ergänzungen diesen Fehler beheben. Gleich zu Anfang wird bei CASE -1 die Versionsnummer ausgegeben:

```
CASE -1
  PRINT "Version 3.0"
```

Fügen Sie hinter den PRINT-Text noch die Angabe des Datenkanals für das Ausgabe-Fenster an und gleich hinter dieser Zeile den Befehl RELSEEK zum Zurücksetzen des Zeigers, so daß der Text auf dem Bildschirm erscheint:

```
CASE -1
  PRINT #1,"Version 3.0"
  RELSEEK #1,0
```

Damit ist auch diese Ausgabe funktionsfähig.

Ein schwerwiegenderer Fehler befindet sich zwischen den folgenden Zeilen der Eingabe-Auswertung. Die Abfrage des Menüpunktes Speicherauswahl (Memory) ist leider um eine Stelle verrückt.

Die CASE-Anweisung hat in der uns vorliegenden Shell die Nummer -22. Damit wird aber die Trennleiste im Menü abgefragt, die man nicht auswählen kann. Korrigieren Sie die Nummer in eine -23 und schon läuft das Programm und besonders diese Funktion ohne Probleme.

Verbesserungen

Gleich das richtige Verzeichnis!

Da Sie sicher immer mit dem gleichen Datenverzeichnis arbeiten werden, ist es eigentlich unsinnig, den Quelltext der Shell so allgemein formuliert zu lassen. Warum soll man nicht seine Standard-Einstellungen übernehmen.

Dies ist besonders empfehlenswert, wenn Sie die Shell von der Workbench aus starten möchten, da hier - wie schon ganz zu Anfang erwähnt wurde - leider jeder Bezug zu einem Verzeichnis fehlt und einfach die Diskette im Laufwerk 0 angesprochen wird.

Für eine Abhilfe reicht es nun aus, einfach die Zeile im Hauptprogramm zu verändern, in der sie folgenden Text finden:

```
'
' Aktueller Pfad
p$=DIR$(0)
'
```

Tragen Sie dafür einfach in die Zuweisungszeile anstelle von DIR$(0) den von Ihnen gewählten Pfad ein. Das könnte die Programmdiskette sein:

```
p$="GFA-Workdisk:Programme"
```

oder auch das Festplattenverzeichnis

```
p$="DH0:GFA/Programme"
```

Damit ist dieses Problem schon gelöst. Compilieren Sie einfach die Shell neu und haben Sie viel Spaß bei der Arbeit.

Standard-Pfad auch für Kommandos

Die GFA-Commandos GFA_BCOM und GL werden immer im aktuellen Verzeichnis vermutet. Dies mag beim Diskettenbetrieb zwar der Fall sein, aber spätestens mit verschiedenen Disketten oder einer Festplatte wird es zu Qual. Man möchte auch hierfür eine Regelung treffen.

Diese Pfad-Einstellung können Sie ebenfalls am Anfang des Shell-Listings vornehmen. Folgende Zeilen sind zu ändern:

```
' Programmnamen
gfaint$="GFABasic"
gfacom$="GFA_BCOM"
gfalnk$="GL"
'
```

Tragen Sie z.B. "C:...." davor ein, dann werden alle Kommandos wie CLI-Kommandos verstanden. NAtürlich müssen die Befehle auch im C-Verzeichnis zu finden sein.

Sie können aber auch den Interpreter von einer anderen Diskette holen, weil sie vielleich keinen Platz mehr hatten. Das sieht dann so aus:

```
' Programmnamen
gfaint$="GFABASIC:GFABasic"
gfacom$=":GFA_BCOM"
gfalnk$=":GL"
'
```

Auch hier sollten sich die Einstellungen ganz nach den persönlichen Bedürfnissen richten. Vergessen Sie aber danach nicht, den Programmtext zu compilieren, sonst haben sie nur wenig von Ihren Verbesserungen!

Druckerprotokoll

Möchten Sie die Texte, die im Ausgabe-Fenster der Compiler-Shell erscheinen für länger festhalten, gibt es eine ganz einfache Möglichkeit, dies zu tun.

Da es sich bei dem Fenster nicht um ein Standard-Fenster handelt, das über ein OPENW geöffnet wurde, sonder dieses als Datenkanal existiert, reicht es aus, die Kanalbezeichnung zu ändern. Die vorher so lautende Zeile:

```
sss$="CON:0/"+STR$(INT(maxy&/2))+"/640/"+STR$(INT(maxy&/2))+"/Ausgabe:"
```

heißt danach z.B. so:

```
sss$="PRT:"
```

Damit werden alle Zeilen auf dem Drucker ausgeben und können später untersucht werden, wenn ein Programm sehr viele Fehler hatte, oder die Ausgabe zu schnell lief.

Sie können die Daten natürlich auch in ein File schreiben. Voraussetzung dafür ist allerdings, daß genügend Speicher auf der Diskette oder Festplatte vorhanden ist, da sonst Probleme auftreten, die die Compilierung aufhalten. Die Zeile könnte so lauten:

```
sss$="DH0:Compiler-Protokoll"
```

Multitasking bei Execute

Stört es Sie auch, daß Programme zwar von der Shell gestartet werden können, aber diese dann so lange nicht mehr reagiert, bis das Programm abgelaufen ist? Dieses Problem macht sich besonders bemerkbar, wenn das Compilat einen Fehler hat und sich nicht mehr zurückmeldet. Weder das Programm noch die Shell kann dann bedient werden. Ein Reset wird nötig.

Dabei kann mit einer klitzekleinen Änderung das Multitasking eingeführt werden. Suchen Sie sich dazu in der Prozedur Check die Abfrage nach der Taste "T":

```
CASE -9,20 !^T
  PRINT #1,"Executing ";tprg$
  RELSEEK #1,0        !clear buffer
  t%=TIMER
  e%=EXEC(tprg$,0,0)
  tmx
```

Hierbei reicht es, wenn Sie die vorletzte Zeile wie folgt ergänzen:

```
e%=EXEC("RUN"+tprg$,0,0)
```

Der CLI-Aufruf wird damit nämlich ins Multitasking gelegt und die Shell meldet sich sofort zurück.

Es gibt dabei nur einen - wie ich meine - verschmerzbaren Schönheitsfehler; die Zeitmessung arbeitet nicht mehr ordnungsgemäß. Dies können Sie aber leicht beheben, indem Sie das entsprechende Programm mit Execute starten.

18. Vektor- und Matrizenberechnungen

Auf den folgenden Seiten werden die neu hinzugekommenen Befehle der Version 3.5 erläutert. An den alten Befehlen hat sich grundsätzlich nichts geändert. Es wurden kleinere Fehler beseitigt und auch einige Ausführungsgeschwindigkeiten verbessert. Für den Programmierer hat sich damit aber nichts Bedeutendes getan. Allein die neuen Befehle sind es, die ihn interessieren werden.

Einleitend wollen wir mit der Frage beginnen, was überhaupt Vektoren und Matrizen - der Titel dieses Kapitels - sind. Schließlich stammen diese Begriffe aus der Mathematik und nicht jeder läuft mit dem Wissen eines Mathematikstudenten herum.

Um so ärgerlicher wird derjenige reagieren, dem nun gesagt werden muß, daß man als Vektor letztendlich alles auffassen kann, was sich nur in Zahlen kleiden läßt, und daß Matrizen vereinfachend gesehen nur eine Ansammlung von Vektoren sind. Da diese "hochmathematische" Definition zwar sehr komplex und zugleich allgemein gehalten ist, können wir praktisch noch nichts damit anfangen. Erst wenn man ein Beispiel einführt, werden Ihre Augen zu leuchten beginnen!

Vektoren sind Zahlen, Paare, Tripel oder allgemein n-Tupel. Das heißt, eine beliebige, aber festgelegte Anzahl von Zahlen ist ein Vektor. Einige Beispiele:

1. Beispiel

Die Zahl 253 ist ein Vektor; auch die Zahl 7 ist ein Vektor.

Jede Zahl r ist ein Vektor. Dabei kann r sowohl eine reelle - eine Zahl mit Nachkommastellen - als auch natürliche - also ganze - Zahl sein.

2. Beispiel

Das Zahlenpaar 5, 7 ist *ein* Vektor; das Zahlenpaar 2557, 964 ist auch ein Vektor.

Allgemein gesprochen ist jedes Zahlenpaar r, s ein Vektor. Für die Variablen gilt auch das oben Gesagte.

3. Beispiel

Jede Anzahl von Zahlen ist ein Vektor. Allerdings können wir nur Tupel (so nennt man eine beliebige Anzahl von zusammengehörenden Zahlen) gleichen Grades vergleichen, d.h. die Anzahl der Zahlen, die sich zu einem Vektor zusammenfassen, muß immer gleich sein.

Vektoren als grafische Punkte

Nun kommt das Geniale daran! Wir können z.B. den Bildschirm als eine endlich große Anzahl von Vektoren auffassen. Die Zahlenpaare bilden sich hierbei aus der X- und Y-Koordinate. Jeder Vektor auf dem Bildschirm hat nun die folgende Form:

```
V=(x,y)
```

Dabei meinen wir aber nicht nur den Punkt, der zu dieser Koordinate führt, sondern auch gleichzeitig den Strahl, der vom Koordinatenursprung zu diesem Punkt führt. Ein Vektor ist also ein Strahl oder Pfeil mit dem Ursprung des Koordinatensystems als Anfangspunkt und den Vektorkoordinaten als Endpunkt. Den bezeichnet man als Ortsvektor: Der Vektor zeigt zum Ort des Punktes.

Ebenso gibt es Verbindungsvektoren; sie zeigen von einem Punkt zu einem anderen. Man verwendet sie z.B. bei Gitternetz-Modellen.

Zur Praxis:

Als Voraussetzung für die Vektorrechnung werden die Vektorwerte in ein- oder zweidimensionalen Feldern gespeichert. Wir benutzen also zum Beispiel

```
DIM a(2)
```

für einen Vektor mit zwei Koordinaten. Darin läßt sich dann für unsere Arbeit ein Bildschirmpunkt speichern.

Klärung von Fachbegriffen

Matrix

Als Matrix bezeichnet man jedes Feld, das zur Speicherung von Zahlen verwendet wird. Es gibt zwei- und dreidimensionale Matrizen (so der Plural). Viele Anwendungen werden aber mit zweidimensionalen Matrizen berechnet. Mehr als drei Dimensionen sind zwar theoretisch bei einer Matrix möglich, entziehen sich aber der Anschaulichkeit und werden von den meisten Befehlen auch nicht mehr unterstützt.

Vektor

Ein Vektor ist eine eindimensionale Matrix. Das heißt, wir haben es mit einer "Reihe" von Zahlen zu tun, die zusammengehören, wie z.B. die Bildschirmkoordinaten. x, y und vielleicht z bilden einen Vektor.

Die Befehle im einzelnen:

Alle Befehle tragen das Schlüsselwort MAT am Anfang!

18.1 Grundbefehle zur Matrizenhandhabung

MAT BASE { M B } — Index-Offset festlegen

```
MAT BASE 0
MAT BASE 1
```

Dieser Befehl bestimmt wie auch OPTION BASE für Felddimensionen die Dimension einer Matrix oder eines Vektors. Es wird der Startwert festgelegt, bei dem GFA-BASIC die Feld-Indizes verwaltet.

Voreingestellt ist hier MAT BASE 1, was auch der mathematischen Grundlage und Vereinbarung entspricht.

Es ist nur sinnvoll, diesen Befehl zu verwenden, wenn man vorher mit OPTION BASE 0 den Startindex für Felder verändert hat. Dann paßt man hiermit die Matrizenverwaltung den Gegebenheiten an.

Der Standard-Wert für MAT BASE ist 1!

MAT CLR { M CL } — Eine Matrix löschen

```
MAT CLR m()
```

Um ein Feld schnell und zügig zu löschen, empfiehlt sich dieser Befehl. Er setzt unabhängig von der Dimensionierung alle Elemente des Feldes auf Null.

Gleiches können Sie auch über ARRAYFILL m(),0 erreichen. Die Befehle führen die gleiche Operation aus. Dieser wurde nur geschaffen, damit er sich in die Gruppe der Matrizenbefehle eingliedert.

Sinnvoll ist die Anwendung z.B. bei Feldern, bei denen bei großer Dimension nur weniger Felder Werte enthalten. Diese setzt man dann explizit.

MAT SET { M SE } — Eine Matrix auf einen Wert setzen

```
MAT SET m()=x
```

Dieser Befehl arbeitet ähnlich dem obigen, nur daß die Matrix nicht auf Null, sondern auf einen beliebigen Wert gesetzt wird. Auch hier ist wieder der Bezug zum Befehl ARRAYFILL m(),x.

Nützliche Funktion erfüllt dieser Befehl ebenfalls bei Matrizen, die viele gleiche Werte enthalten. Nur anderslautende werden dann noch eingetragen. Man erspart sich damit viel Programmierarbeit und Dateneingabe.

MAT ONE { M O } — Eine Einheitsmatrix erzeugen

```
MAT ONE m()
```

Auch dieser Matrizen-Befehl gehört zu denjenigen, die in die Matrix Werte schreiben. Hierbei wird allerdings eine Einheitsmatrix erzeugt.

Mit einer Einheitsmatrix bezeichnet man Matrizen, die fast vollständig auf Null gesetzt sind, außer den Elementen mit paarweise gleichem Index (1,1), (2,2), (3,3)... Diese enthalten den Wert 1.

Hinweis: Beachten Sie, daß die Matrix quadratisch sein muß. Das heißt, die Matrix muß zweidimensional sein (ein Feld mit zwei Indizes) und ihre Indizes müssen die gleiche Mächtigkeit besitzen.

18.2 Ein- und Ausgabe der Matrizendaten

MAT READ { M R } — Eine Matrix aus DATAs einlesen

```
MAT READ m()
```

Zur Vereinfachung der Dateneingabe in eine Matrix können vorgegebene Matrizen in DATA-Zeilen abgespeichert sein. Um sich die sehr häufig vorkommende Arbeit des Einlesens der Daten zu ersparen, wurde ein Befehl in die Sammlung der Matrizen-Befehle aufgenommen, der dies in einem Atemzug erledigt.

MAT READ liest ab der bestehenden Position des DATA-Zeigers soviele Daten, wie für das angegebene Feld nötig sind. Je nachdem also, ob es ein ein- oder mehrdimensionales Feld ist, werden entsprechend Spalten und Zeilen gelesen.

Die Anzahl ist von der durch MAT BASE und in DIM festgelegten Feld-Dimension abhängig.

Hinweis: Es war in der uns vorliegenden Version nicht möglich, mehrere Felder bzw. Vektoren/Matrizen gleichzeitig hintereinander einzulesen, wie es bei READ möglich ist.

Für Beispiele schauen Sie in eines der folgenden Beispielprogramme. Wir werden diesen Befehl immer wieder verwenden, um Beispieldaten aus DATA-Zeilen einzulesen.

MAT PRINT { M P oder M ? } — Eine Matrix ausgeben

```
MAT PRINT [#c,]m()[,g,n]
```

Zur einfachen Ausgabe einer Matrix wurde dieser Befehl geschaffen, der genauso wie der allseits bekannte PRINT-Befehl arbeitet. Es entfällt hier, genau wie bei MAT READ, die lästige

Schleife, die nötig wäre, um alle Elemente auszugeben. MAT PRINT berechnet automatisch die Anzahl der Zeilen und Spalten.

Zusätzlich können noch Angaben über das Format der Zahlen gemacht werden. Mit g und n sind diese Einstellungen zu beeinflussen: g bestimmt die gesamte Breite der Zahlendarstellung und n die Anzahl der Nachkommastellen (die Formatierungsmethode wurde aus Pascal übernommen und wurde schon bei der Funktion STR$() verwendet).

g bestimmt die gesamte Breite in Zeichen. Das bedeutet, daß für die Zahl vor dem Komma genau (g-n-1) Zeichen freibleiben (abzüglich der Nachkommastellen und dem Komma selbst).

Weiterhin kann der MAT PRINT-Befehl durch eine Dateikennung ergänzt werden. Die Ausgabe erfolgt dann nicht auf dem Standard-Bildschirm, sondern auf dem Datenkanal. Dies kann einerseits eine Datei sein, andererseits aber auch eine andere Bildschirm-Ausgabe (CON:) bzw. der Drucker!

Für Anwendungsbeispiele schauen Sie bitte in den nachfolgenden Programmen nach. Auch diesen Befehl werden wir immer wieder zur Kontrolle und zu Demonstrationszwecken verwenden.

MAT INPUT { M I } Eine Matrix aus einer Datei einlesen

```
MAT INPUT #c,m()
```

Genauso wie es möglich ist, eine Matrix aus DATA-Zeilen einzulesen, können die Daten auch aus einer Datei geholt werden. Dies erscheint besonders plausibel, wenn man bedenkt, daß mit MAT PRINT die Daten ja auch in eine Datei geschrieben werden können (siehe dort).

Als einzige Angaben werden die Kanalnummer des Ausgabe-Kanals und die Feldbezeichnung angegeben.

Es ist hierbei nicht zulässig, nur Teilbereiche des Feldes einzulesen. Achten Sie auch darauf, daß noch genügend Daten lesbar sind, da sonst eine Fehlermeldung auftritt.

Beispielprogramm:

```
' Beispiel: MAT I/O
'
' Speichern und Ausgeben von vorhandenen Matrizen
'
' Programm zur Demonstration der
' Vektor- und Matrizen-Befehle
'
' (c) 1990 by DATA BECKER
' (p) 1990 by Wgb
'
OPTION BASE 0
MAT BASE 1
'
DIM v(3),m(2,5)
'
MAT READ v()
MAT READ m()
'
PRINT "Die gelesenen Werte der Matrizen ..."
MAT PRINT v(),5,1
PRINT
MAT PRINT m(),5,1
PRINT
'
PRINT "Die Daten werden geschrieben ..."
OPEN "o",#1,"ram:Matrizen"
MAT PRINT #1,v()
MAT PRINT #1,m()
CLOSE #1
PRINT
'
PRINT "Matrix m() und v() werden verändert ..."
MAT SET v()=5
MAT SUB m(),INT(RND(1)*5)
MAT PRINT m(),5,1
PRINT
'
PRINT "Matrix v() und m() werden wieder eingelesen ..."
OPEN "i",#1,"ram:Matrizen"
MAT INPUT #1,v()
MAT INPUT #1,m()
CLOSE #1
MAT PRINT v(),5,1
PRINT
MAT PRINT m(),5,1
```

```
PRINT
'
'
' DATA-Zeilen
' Vektor
DATA 7,5,2
' Matrix
DATA 1,2,7,3,4
DATA 9,4,1,8,3
```

MAT CPY { M C } — Eine Matrix kopieren

```
MAT CPY a([i,j])=b([k,l])[,h,w]
```

Dieser Befehl kopiert eine bestehende Matrix in eine andere hinein. Dabei kann man damit sowohl eine Matrix duplizieren, d.h. ein genaues Abbild der vorhandenen Matrix schaffen, als auch einen Teilbereich in eine größere Matrix einsetzen.

1. Eine Matrix duplizieren

Benötigt man für Nebenrechnungen eine Arbeitsmatrix, ist es ratsam, den Inhalt der Ausgangsmatrix in ein separates Feld zu kopieren. Dafür eignet sich dieser Befehl. Mit der Zeile

```
MAT CPY n()=a()
```

enthält das Feld n(), das in diesem Fall die gleichen Dimensionen besitzen wie die Matrix a() muß, deren Inhalt. Man kann diesen Befehl hier wie eine einfache Variablenzuweisung auffassen.

Sollte die Ziel- oder Ausgangsmatrix andere Indexmaxima besitzen, werden nur die Elemente kopiert, die in beiden Matrizen die gleichen Indizes besitzen. Dies gilt auch für alle weiteren Varianten!

2. Eine Zeile in eine andere Matrix kopieren

Um nur eine Zeile einer Matrix in eine andere zu kopieren, benötigt man weitere Angaben. Wesentlich ist die Angabe der Zeile, die kopiert werden soll. Man gibt dafür die Eckkoordinaten der linken oberen Ecke an, also z.B. (1,1) für die oberste linke Ecke einer Matrix. Außerdem benötigt GFA-BASIC die Anzahl der Zeilen, also eins, und die Anzahl der Elemente in der Zeile, in unserem Beispiel (s.u.) 5. Damit ist der Quellbereich definiert. Nun fehlt nur noch der Zielbereich, den wir auch als Koordinaten in der Matrix definieren.

Sehen Sie hier unser kleines Beispiel:

```
' Zeile kopieren
'
DIM a(5,5),b(5,5)
'
MAT SET a()=3
MAT SET b()=2
'
MAT PRINT a()
PRINT
MAT PRINT b()
PRINT
'
MAT CPY a(2,1)=b(1,1),1,5    ! von b nach a
'
MAT PRINT a()
PRINT
```

Beachten Sie hierbei die Angaben beim Kopierbefehl. a(2,1) bedeutet, daß die Daten in der zweiten Zeile ab dem ersten Element abgelegt werden.

b(1,1) setzt die Quellposition auf die erste Zeile und erste Spalte im Feld b(). 1,5 heißt, daß eine Zeile und genau fünf Elemente (die Breite der Matrix) kopiert werden.

3. Eine Spalte kopieren

Um eine Spalte zu kopieren, müssen wir ähnlich wie beim Kopieren von Zeilen verfahren. Es ändern sich lediglich die am

Ende angefügten Parameter. Sie geben bei den Zeilen das Maximum des definierten Feldes an und bei den Spalten eine Eins. Sehen Sie dazu ein ähnliches Beispiel wie oben:

```
' Spalte kopieren
'
DIM a(5,5),b(5,5)
'
MAT SET a()=8
MAT SET b()=1
'
MAT PRINT a()
PRINT
MAT PRINT b()
PRINT
'
MAT CPY a(1,2)=b(1,1),5,1     ! von b nach a
'
MAT PRINT a()
PRINT
```

4. Einen rechteckigen Ausschnitt kopieren

Hierfür verwenden wir die oben schon benutzen Parameter und beschränken uns nicht auf eine Zeile oder eine Spalte, sondern nutzen einen rechteckigen Bereich.

Die Angaben beim Feld auf der linken Seite des Gleichheitszeichens geben immer noch die linke obere Ecke im Zielfeld an (Zeile, Spalte). Die Angaben im Feld auf der rechten Seite stehen für die Quelldaten (auch hier Zeile, Spalte).

Allein die Werte hinter dem Feld für die Quelldaten werden nun vollständig ausgenutzt. Auch hier bezeichnet die erste Zahl die Anzahl der Zeilen und die zweite die Anzahl der Spalten. Das nachfolgende Kurzprogramm verdeutlicht dies:

```
' Bereich kopieren
'
DIM a(5,5),b(5,5)
'
MAT SET a()=0
MAT SET b()=1
'
MAT PRINT a()
PRINT
```

```
MAT PRINT b()
PRINT
'
MAT CPY a(2,2)=b(1,1),3,3   ! von b nach a
'
MAT PRINT a()
PRINT
```

MAT XCPY { M X } Kopiert eine Matrix transponiert

```
MAT XCPY a([i,j])=b([k,l])[,h,w]
```

Eine Variante des Matrizen-Kopierbefehls liegt mit diesem Befehl vor. Er kopiert mit den anderen Möglichkeiten, denn es werden hierbei Zeilen und Spalten vertauscht.

Dadurch können Sie Spalten-Vektoren in Zeilen-Vektoren und umgekehrt umwandeln. Dies eignet sich besonders für die Einfügung in Matrizen, die anders organisiert sind.

1. Eine Zeile in eine Spalte kopieren

Die Parameterangaben des XCPY-Befehls sind genau die gleichen wie die der normalen CPY-Version. Verändern Sie für einen einfachen Test eines der obigen Beispiele, indem Sie vor das Befehlswort noch den Buchstaben X setzen. Im Ergebnis wird sich zeigen, daß eine Zeile (z.B. mit den Angaben ,1,5) nun in eine Spalte kopiert wird.

Die Parameter in der Zielmatrix a() geben dabei immer noch die linke obere Ecke an. Auch die Koordinaten aus der Quellmatrix stehen für die linke obere Ecke, ebenso wie die Werte für Zeilenzahl und Spaltenelemente. Verändert ist hierbei nur der Eintragung in die Zielmatrix: senkrecht statt waagerecht und waagerecht statt senkrecht.

2. Einen Bereich spiegeln

Um einen Bereich zu spiegeln, reichen ebenfalls die Standard-Angaben aus. Leicht demonstrieren läßt sich dies an einem Beispielprogramm, das mit einem Feld arbeitet, in dem jedes Element den Wert seiner Position enthält. Sehen Sie dafür einfach unser nachfolgendes Beispiel:

```
DIM a(5,5),b(5,5)
MAT SET a()=0
FOR i=1 TO 5
  FOR j=1 TO 5
    INC z
    b(i,j)=z
  NEXT j
NEXT i
'
MAT PRINT a(),2,0
PRINT
MAT PRINT b(),2,0
PRINT
'
MAT XCPY a(2,2)=b(1,1),3,3
'
MAT PRINT a(),2,0
PRINT
```

Die Spiegelachse verläuft bei diesen Kopiervorgang diagonal von links oben nach rechts unten.

MAT TRANS { M T } Kopiert und transponiert eine Matrix

```
MAT TRANS a()[=b()]
```

Auch hierbei handelt es sich um einen Kopierbefehl, der aber anders als die beiden vorgehenden arbeitet, weil er nicht die Spalten- und Zeileneinteilung beachtet. So ist es mit MAT TRANS möglich, eine vierspaltige und dreizeilige Matrix in eine dreispaltige und vierzeilige umzukopieren.

Dies ist auch gleich die Bedingung für den Befehl. Wendet man MAT TRANS auf zwei Matrizen an, muß die erste als Zeilen-

zahl die Anzahl der Spalten der zweiten Matrix besitzen. Gleiches gilt für die Spaltenzahl der ersten Matrix.

Handelt es sich um eine quadratische Matrix, kann der Befehl auf sie selbst angewendet werden, da ja beide Bedingungen erfüllt sind.

Das Ergebnis dieses Befehls können Sie am einfachsten am folgenden Beispiel erkennen. Es erzeugt eine Matrix, deren Zeilen gleiche Werte haben. Da diese Matrix quadratisch ist, kann sie "in sich" kopiert werden.

```
DIM a(8,8)
FOR i=1 TO 8
  FOR j=1 TO 8
    a(i,j)=i
  NEXT j
NEXT i
MAT PRINT a(),2,0
PRINT
MAT TRANS a()
MAT PRINT a(),2,0
PRINT
 1, 1, 1, 1, 1, 1, 1, 1
 2, 2, 2, 2, 2, 2, 2, 2
 3, 3, 3, 3, 3, 3, 3, 3
 4, 4, 4, 4, 4, 4, 4, 4
 5, 5, 5, 5, 5, 5, 5, 5
 6, 6, 6, 6, 6, 6, 6, 6
 7, 7, 7, 7, 7, 7, 7, 7
 8, 8, 8, 8, 8, 8, 8, 8
 1, 2, 3, 4, 5, 6, 7, 8
 1, 2, 3, 4, 5, 6, 7, 8
 1, 2, 3, 4, 5, 6, 7, 8
 1, 2, 3, 4, 5, 6, 7, 8
 1, 2, 3, 4, 5, 6, 7, 8
 1, 2, 3, 4, 5, 6, 7, 8
 1, 2, 3, 4, 5, 6, 7, 8
 1, 2, 3, 4, 5, 6, 7, 8
```

18.3 Rechnen mit Matrizen

Kommen wir nun zu den Rechenbefehlen für Matrizen. Sie bilden einen wichtigen Bestandteil der Matrizen-Befehle. Mit ihnen können die Werte ganzer Felder (so man sie im herkömmlichen

Sinne betrachtet) auf einmal addiert oder subtrahiert werden. Von hohem rechnerischen Interesse sind aber auch Produktbildungen, mit denen innerhalb der Matrizenrechnung recht komplizierte, aber auch mächtige Operationen durchgeführt werden können.

MAT ADD { M A } — Addiert zwei Matrizen

```
MAT ADD a()=b()+c()
MAT ADD a(),b()
MAT ADD a(),x
```

Mit MAT ADD können zwei Vektoren oder Matrizen addiert werden. Das Ergebnis liegt dabei elementweise vor, d.h. das erste Element der ersten Matrix wurde zum ersten Element der zweiten Matrix addiert usw. Die Funktion des Befehls ist also sehr einfach zu verstehen.

In der ersten Befehlsvariante schreibt man die Addition wie eine Wertzuweisung von normalen Variablen. MAT ADD weist danach der Matrix a das Ergebnis der Elementaddition von b und c zu. In der zweiten Variante werden auch a und b addiert. Allerdings steht hier das Ergebnis wiederum in a. Damit geht der alte Inhalt der Matrix a verloren (Sie können ihn natürlich durch Subtraktion von b wiedererhalten). Beachten Sie dies, wenn Sie auf die Werte später wieder zurückgreifen wollen. Die letzte Befehlsvariante addiert zu jedem der Elemente der Matrix a den festen Wert x. Es wird z.B. allen Einträgen 7 dazuaddiert.

Beachten Sie, daß die Addition nur für Vektoren und Matrizen gleicher Ordnung definiert ist. Das bedeutet, Sie können nur Felder mit gleichen Dimensionen verwenden. Im Einzelfall müßten Sie zuvor mit MAT CPY, MAT XCPY oder MAT TRANS die Matrix in eine Matrix der benötigten Größe überführen.

Interessant für diesen Befehl sind noch die Abkürzungen. Es existiert nicht nur die oben im Kopf genannte Möglichkeit von { M A }, sondern es gibt auch einen Spezialfall: Für die erste Variante reicht es, { M a()=b()+c() } zu schreiben, denn der Interpreter erkennt selbständig, daß es sich um eine Matrizen-Addition handelt.

Das folgende Beispielprogramm soll die Arbeit dieses Befehls verdeutlichen:

```
' Beispiel: MAT ADD
'
' Konstantenaddition
' Feldaddition mit Datenverlust
' Feldaddition ohne Datenverlust
'
' Programm zur Demonstration der
' Vektor- und Matrizen-Befehle
'
' (c) 1990 by DATA BECKER
' (p) 1990 by Wgb
'
OPTION BASE 0             ! Setzt den Beginn der Feldindizes auf 1
MAT BASE 1                ! s.o. für Matrizen
'
DIM a(3,4),b(3,4),c(3,4)
MAT READ a()
MAT READ b()
'
PRINT "Feld a(), Urzustand"
MAT PRINT a(),5,1
PRINT
'
PRINT "Konstantenaddition x=1"
'
MAT ADD a(),1                   ! Dies ist die Beispielzeile
PRINT "Feld a(), nach der Addition"
MAT PRINT a(),5,1
PRINT
'
PRINT "1. Feldaddition Feld b() ist Invers zu a()"
'
MAT ADD a(),b()                 ! Dies ist die Beispielzeile
PRINT "Feld a() nach der Addition"
MAT PRINT a(),5,1
PRINT
'
PRINT "2. Feldaddition Feld b() zu sich selbst addiert in c()"
'
MAT ADD c()=b()+b()             ! Dies ist die Beispielzeile
```

```
MAT PRINT c(),5,1
PRINT
'
' DATA-Zeilen
DATA 1,2,3,4,5,6,7,8,9,10,11,12
DATA -2,-3,-4,-5,-6,-7,-8,-9,-10,-11,-12,-13
```

MAT SUB { M S } — Subtrahiert zwei Matrizen

```
MAT SUB a()=b()-c()
MAT SUB a(),b()
MAT SUB a(),x
```

Der Befehl MAT SUB dient zum Subtrahieren zweier Vektoren oder Matrizen. Die Elemente werden dabei einzeln miteinander verknüpft. Das Ergebnis wird wie auch bei MAT ADD gehandhabt. Das bedeutet für die erste Variante, daß das Ergebnis in a() gespeichert wird. Beim zweiten Befehl wiederum in a(), dort muß aber beachtet werden, daß die Daten verlorengehen., Und bei der letzten Möglichkeit werden grundsätzlich die Daten im Feld a() verändert.

Achten Sie darauf, daß dieser Befehl nur für Matrizen und Vektoren gleicher Ordnung definiert ist.

Zur Schreibweise innerhalb von Programmen: Die Abkürzung lautet { M S } und kann bei der ersten Befehlsvariante auch { M a()=b()-c() } heißen, da der Interpreter den Befehl am Rechenzeichen erkennt.

Beispielprogramm:

Das folgende Beispielprogramm führt ähnliche Operationen durch, wie es auch das Programm zur Erläuterung des Befehls MAT ADD tat. Sie können daran die Funktion der Befehle sehr leicht erkennen, indem Sie beide Listings miteinander vergleichen. Beachten Sie allerdings, daß die Daten am Ende geändert wurden, damit auch das vorletzte Beispiel funktioniert!

```
' Beispiel: MAT SUB
'
' Konstantensubtraktion
' Feldsubtraktion mit Datenverlust
' Feldsubtraktion ohne Datenverlust
'
' Programm zur Demonstration der
' Vektor- und Matrizen-Befehle
'
' (c) 1990 by DATA BECKER
' (p) 1990 by Wgb
'
OPTION BASE 0           ! Setzt den Beginn der Feldindizes auf 1
MAT BASE 1              ! s.o. für Matrizen
'
DIM a(3,4),b(3,4),c(3,4)
MAT READ a()
MAT READ b()
'
PRINT "Feld a(), Urzustand"
MAT PRINT a(),5,1
PRINT
'
PRINT "Konstantensubtraktion x=1"
'
MAT SUB a(),1                     ! Dies ist die Beispielzeile
PRINT "Feld a(), nach der Subtraktion"
MAT PRINT a(),5,1
PRINT
'
PRINT "1. Feldsubtraktion Feld b() ist invers zu a()"
'
MAT SUB a(),b()                   ! Dies ist die Beispielzeile
PRINT "Feld a() nach der Subtraktion"
MAT PRINT a(),5,1
PRINT
'
PRINT "2. Feldsubtraktion Feld b() von sich selbst subtrahiert in
c()"
'
MAT SUB c()=b()-b()               ! Dies ist die Beispielzeile
MAT PRINT c(),5,1
PRINT
'
' DATA-Zeilen
DATA 1,2,3,4,5,6,7,8,9,10,11,12
DATA 0,1,2,3,4,5,6,7,8,9,10,11
```

MAT MUL { M M } — Multipliziert zwei Matrizen

```
MAT MUL a()=b()*c()
MAT MUL x=a()*b()
MAT MUL x=a()*b()*c()
MAT MUL a(),x
```

Für die Multiplikation von Matrizen in den hier beschriebenen Befehlen zuständig. Eine Matrix mit einer anderen zu multiplizieren, bedeutet, jedes Element mit jedem einzeln zu multiplizieren und die Summe aller Ergebnisse für ein Element an dessen Position als Ergebnis auszugeben. Da dies etwas komplizierter wird, sollen einige Beispiele das verdeutlichen:

Wir haben die folgenden Matrizen:

```
s(2,2)
0.5    0.2
0.5    0.8
```

und

```
m(4,2)
1      0
0      1
-1     0
0      -1
```

Multipliziert man nun die erste mit den zweiten unter Anwendung des folgenden Befehls

```
MAT MUL b()=m()*s()
```

so ergibt sich folgendes Ergebnis:

```
b(4,2)
0.5    0.2
0.5    0.8
-0.5   -0.2
-0.5   -0.8
```

Erwähnenswert ist, daß die Ergebnismatrix die Dimensionierung besitzt, die sich aus der Anzahl der Spalten der zweiten Matrix und der Anzahl der Zeilen der ersten Matrix zusammensetzt. Aber wie kommt man nun zu diesem Ergebnis? Hier ist die Lösung:

```
0.5*1+0.5*0=0.5          0.2*1+0.8*0=0.2
0.5*0+0.5*1=0.5          0.2*0+0.8*1=0.8
0.5*-1+0.5*0=-0.5        0.2*-1+0.8*0=-0.2
0.5*0+0.5*-1=-0.5        0.2*0+0.8*-1=-0.8
```

Allgemeine Regeln für die Matrizenmultiplikation:

Damit zwei Matrizen miteinander multipliziert werden können, muß die Spaltenzahl der einen mit der Zeilenzahl der anderen übereinstimmen.

Zur Multiplikation verwendet man folgendes Verfahren: Um die Zahl in der i-ten Zeile und j-ten Spalte der Ergebnismatrix zu erhalten, multipliziert man je die Zahl der i-ten Zeile aus der ersten Matrix mit den entsprechenden Zahlen der j-ten Spalte der zweiten Matrix. Alle Ergebnisse addiert man und hat damit die gesuchte Zahl. So verfährt man mit allen Elementen der Matrix.

Beispielprogramm:

```
' Beispiel: MAT MUL
'
' Abbildungen
'
' Programm zur Demonstration der
' Vektor- und Matrizen-Befehle
'
' (c) 1990 by DATA BECKER
' (p) 1990 by Wgb
'
'
' Variablen
dimension=2
anzahl=4
xdarst=100
ydarst=100
xoff=320
yoff=128
'
```

```
' Vektoren
DIM p(anzahl,dimension),b(anzahl,dimension)
'
' Abbildungsmatrix
DIM m(dimension,dimension)
'
' Daten einlesen: Matrix ... Punkte
MAT READ m()
MAT READ p()
'
' Daten ausgeben
'
PRINT "Abbildungsmatrix"
MAT PRINT m(),5,2
PRINT
PRINT "Punkte ..."
MAT PRINT p(),5,2
PRINT
'
COLOR 3
@drawp
'
' Punkte abbilden
'
COLOR 1
FOR i=1 TO 15
  PAUSE 50
  LOCATE 1,12
  MAT PRINT p(),7,4
  MAT MUL p()=p()*m()
  @drawp
NEXT i
END
'
PROCEDURE drawp
  DRAW p(4,1)*xdarst+xoff,yoff-p(4,2)*ydarst
  FOR j=1 TO anzahl
    DRAW  TO p(j,1)*xdarst+xoff,yoff-p(j,2)*ydarst
  NEXT j
RETURN
'
' DATA Zeilen
DATA 0.5,0.2              ! Experimentieren Sie ruhig mit der
DATA 0.5,0.8              ! Abbildungsmatrix (Werte zwischen 0
'                         ! und 1)
'                         ! zentr. Streckung vom Ursprung
'                           1. Element: tend. x-Richtung
'                           2. Element: tend. y-Richtung
'
DATA 1,0                  ! Koordinaten der Urpunkte
DATA 0,1
DATA -1,0
DATA 0,-1
```

Das obenstehende Programm dient zur Abbildung von Punkten (hier als Vektoren). Die Punkte wurden allesamt in einem Feld gespeichert, wodurch die Berechnung der Bildpunkte in einem Rechenschritt durchgeführt werden kann. Außerdem ist es so möglich, beliebig viele Punkte gleichzeitig abzubilden.

Die Abbildungsvorschrift selbst wird aus der Abbildungsmatrix übernommen.

MAT NORM { M NO } — Normiert eine Matrix

```
MAT NORM v(),0
MAT NORM v(),1
```

Matrizen sind nichts anderes als zweidimensionale Vektoren. Unter der Normierung eines Vektors versteht man die Stauchung oder Streckung des Vektors auf die Länge 1. Dabei bleiben Richtung und Vorzeichen des Vektors erhalten.

Der Betrag (die Länge) eines Vektors berechnet sich aus seinem Qudrat:

```
sqr(v²)
```

Dabei ist zu beachten, daß beim Quadrieren eines Vektors eine reelle Zahl als Ergebnis herauskommt. Davon ziehen wir die Wurzel. Die Rechenoperationen heben sich also nicht auf, wie bei der normalen Rechnung mit reellen Zahlen.

Dividiert man nun einen Vektor durch seine Länge, wird er normiert. Wir erhalten den Einheitsvektor.

Das nachstehende Programm führt folgende Arbeiten aus: Zuerst wird der Vektor zusammen mit seiner Länge ausgegeben. Dann wird dieser "per Hand" normiert. Dieses Ergebnis wird ausgegeben und mit dem des vorliegenden Befehls verglichen.

```
' Beispiel: MAT NORM
'
' Vektoren normieren
'
' Programm zur Demonstration der
' Vektor- und Matrizen-Befehle
'
' (c) 1990 by DATA BECKER
' (p) 1990 by Wgb
'
DIM v(10)
'
MAT READ v()
PRINT "Der Vektor ..."
MAT PRINT v(),6,4
'
MAT MUL l=v()*v()
PRINT "besitzt die Länge ";SQR(l)
PRINT
'
MAT MUL v(),1/SQR(l)
PRINT "Der 'von Hand' normierte Vektor ..."
MAT PRINT v(),6,4
MAT MUL l=v()*v()
PRINT "besitzt die Länge ";SQR(l)
PRINT
'
RESTORE
MAT READ v()
MAT NORM v(),1
PRINT "Der 'per Befehl' normierte Vektor ..."
MAT PRINT v(),6,4
MAT MUL l=v()*v()
PRINT "besitzt die Länge ";SQR(l)
PRINT
'
DATA 1,3,1,2,2,3,1,2,1,0
```

Das Verfahren für quadratische Matrizen läufte dabei ebenso ab, wobei hier eine Matrix entweder zeilenweise (man verwendet die Angabe 0 hinter dem Befehl) oder spaltenweise (1) normiert werden kann. Dies hängt von der Auffassung der Zusammensetzung einer Matrix ab. Bei Vektoren ist diese Angabe irrelevant.

MAT DET { M D } Berechnet die Determinante

```
MAT DET y=m([i,j])[,d]
```

Dieser Befehl berechnet die Determinante einer quadratischen Matrix der Ausdehnung d. Bei einer Angabe von i und j wird nur die Determinante eines Teils der quadr. Matrix berechnet.

Die Determinante liefert dabei eine Aussage über die Lösbarkeit der Matrix.

MAT QDET { M QD } Schnellere Determinanten-Berechnung

```
MAT QDET y=m([i,j])[,d]
```

Dieser Befehl liefert im Prinzip das gleiche Ergebnis wie der Befehl MAT DET. Jedoch wird hierbei eine geschwindigkeitsoptimierte Routine verwendet. Es kann also zu leichten Differenzen in der Genauigkeit kommen.

MAT RANG { M RA } Berechnet den Rang einer Matrix

```
MAT RANG y=m([i,j])[,d]
```

Diese Funktion berechnet den Rang einer quadratischen Matrix.

Die Parameter bezeichnen dabei bei Angabe einen Ausschnitt aus der Matrix. i,j geben die linke obere Ecke des Ausschnitts und d die Ausdehnung in x- und y-Richtung an.

MAT INV { M INV } Bestimmt das Inverse einer quadr. Matrix

```
MAT INV i()=m()
```

Das Inverse einer quadratischen Matrix ist definiert als Ergebnis der Multiplikation zweier Matrizen. Dabei muß die Einheitsmatrix herauskommen.

Als Ergebnis steht in i() die zu m() inverse Matrix.

19. Mehr Bedienkomfort

Zusätzlich zu Vektor- und Matrizenbefehlen sind noch ein paar andere hinzugekommen, die wir hier zusammen mit den Ergänzungen und Änderungen des Editors besprechen wollen.

19.1 Mathematische Befehle

FACT { FACT } **Berechnet die Fakultät**

```
f=FACT(n)
```

Funktion zur Berechnung der Fakultät. Damit ist folgendes Rechenverfahren definiert:

```
n! (man liest n-Fakultät)
n! = n*(n-1)!
```

Beispiel:

```
4! = 4*(4-1)! = 4*3*(3-1)! = 4*3*2*(2-1)! = 4*3*2*1 = 24
```

Die Fakultät ist damit das Produkt über die Zahlen 4 bis 1 oder allgemein gesprochen n bis 1.

0! ist definiert als 1, damit dies bei der Multiplikation als das neutrale Element verwendet werden kann (ähnlich wie beim Potenzieren mit Null). Die Funktion ist außerdem nur für ganzzahlige Werte definiert. Das Ergebnis kann ebenfalls nur ganzzahlig sein.

GFA-BASIC ist in der Lage, die Fakuktät bis zu dem Wert 421 zu berechnen. Im Vergleich kann ein normaler Taschenrechner gerade noch den Wert 69 verkraften!

```
PRINT FACT(421)
4.967094384179E+923
```

VARIAT { VARIAT } Berechnet die Anzahl der Variationen

```
n=VARIAT(m,k)
```

Diese Funktion berechnet die Anzahl der Variationen, der angegebenen m Elemente zur k-ten Klasse. Wiederholungen werden dabei ausgeschlossen. Die Berechnung kann man "zu Fuß" mit folgender Formel nachvollziehen:

```
n = (m!)/(m-k)!
```

Aus der Formel ergibt sich logischerweise, daß für die Variablen m und k folgende Beziehung gilt:

```
m >= k
```

Sehen Sie eine Beispieltabelle, die ausgehend von m=10 die ersten zehn Klassen darstellt:

```
 1        10
 2        90
 3       720
 4      5040
 5     30240
 6    151200
 7    604800
 8   1814400
 9   3628800
10   3628800
```

COMBIN {COMBIN} Berechnet Anzahl der Kombinationen

```
n=COMBIN(m,k)
```

Ähnlich wie der Begriff der Variationen bildet auch die Anzahl der Kombinationen einen mathematischen Begriff. Auch dieser Funktion liegt eine Definition zugrunde.

```
n = (m!)/((m-k)!*k!)
```

Beachten Sie ebenfalls, daß hier wiederum die oben genannte Bedingung

```
m >= k
```

gelten muß, damit die Funktion definiert ist.

Sehen Sie abschließend auch hier eine Wertetabelle der ersten zehn Klassen mit m=10:

```
 1    10
 2    45
 3   120
 4   210
 5   252
 6   210
 7   120
 8    45
 9    10
10     1
```

_DATA { _DATA } Operationen mit dem DATA-Zeiger

```
_DATA
_DATA=
```

Diese neu eingeführte Variable ermöglicht unabhängig von RESTORE die Manipulation der DATA-Tabellen.

Die Variable _DATA enthält bei der Abfrage einen Integer-Wert, der die aktuelle Position des DATA-Zeigers wiederspiegelt. Diesen können Sie in einer Variablen zwischenspeichern und ggf. wieder verwenden.

Um die Position des DATA-Zeigers zu manipulieren, reicht es im Programmablauf aus, der Variablen _DATA einen vorher gelesenen Wert zuzuweisen. Damit wird der Lese-Zeiger wieder an die derzeitige Position gesetzt.

Beachten Sie folgendes: Setzen Sie den Zeiger niemals an irgendeine Position. Verwenden Sie immer nur bisher gelesene Werte.

Beispielprogramm:

```
' Beispiel: _DATA
'
' Arbeiten mit dem DATA-Zeiger
'
' Programm zur Demonstration der
' neuen GFA-BASIC-Befehle V3.5
'
' (c) 1990 by DATA BECKER
' (p) 1990 by Wgb
'
DIM d(10)          ! Feld für die DATA-Zeiger-Positionen
'
RESTORE            ! Position des DATA-Zeigers auf den
'                    Anfang der Liste setzen und ausgeben
PRINT _DATA
PRINT
'
PRINT "Liste vorwärts lesen ..."
WHILE _DATA        ! Der Wert _DATA ist beim Erreichen des
  '                  Endes der DATA-Liste gleich Null!
  INC i
  d(i)=_DATA       ! Die Zeigerpositionen werden gespeichert
  READ a
  PRINT a
WEND
PRINT
'
PRINT "Liste rückwärts lesen ..."
FOR j=i TO 1 STEP -1
  _DATA= d(j)
  READ a
  PRINT a
```

```
NEXT j
'
END
'
'
DATA 10,20,30,40,50,60,70,80,90,100
```

19.2 Die neuen Editor-Kommandos

Der GFA-Editor wurde um einige wenige Tastaturkommandos ergänzt, die wiederum beweisen, wie wenig verbesserungsbedürftig ein so komfortabler Editor ist.

Einige Ergänzungen hat die Tab-Taste erfahren. Sie war bisher nur wenig differenziert nutzbar. Das hat sich nun gewaltig geändert.

Folgende Kombinationen sind mit Shift und Ctrl möglich, um schnelle Formatierung besonders von Kommentaren zu ermöglichen:

Tab

Wie gewohnt wird der Cursor auf die nächste Tabulatorposition gesetzt. Der Programmtext wird weiterhin nicht verschoben.

Tab+Ctrl

Hiermit kann man den Cursor auf die links von der aktuellen Position liegende Tabulatorposition setzen.

Tab+Shift-links

Dieser Tastendruck fügt bis zur rechts auf die Cursorposition folgenden Tabulatorstelle Leerzeichen ein. Damit kann man Kommentare auf den nächsten Tabstop ausrichten.

Tab+Shift-rechts

Ein Tastendruck und die Editorfunktion löscht alle Leerzeichen der Zeile, in der sich der Cursor befindet.

Leider waren die letzten beiden Funktionen des Editors bei unserer letzten Testversion noch nicht implementiert. Probieren Sie bei Ihrer Version also aus, ob es schon möglich ist. Die Anwendungsmöglichkeiten sind aber auf jeden Fall für den ständigen Anwender unbestreitbar.

Anhang A: ASCII-Tabelle

Code	Zeichen		Code	Zeichen	Code	Zeichen		Code	Zeichen
0	[CTRL]-[@]		64	@	128	□		192	À
1	[CTRL]-[A]		65	A	129	□	(F1)	193	Á
2	[CTRL]-[B]		66	B	130	□	(F2)	194	Â
3	[CTRL]-[C]	(Break)	67	C	131	□	(F3)	195	Ã
4	[CTRL]-[D]		68	D	132	□	(F4)	196	Ä
5	[CTRL]-[E]		69	E	133	□	(F5)	197	Å
6	[CTRL]-[F]		70	F	134	□	(F6)	198	Æ
7	[CTRL]-[G]	(Beep)	71	G	135	□	(F7)	199	Ç
8	[CTRL]-[H]	(BACKSPACE)	72	H	136	□	(F8)	200	È
9	[CTRL]-[I]	(TAB)	73	I	137	□	(F9)	201	É
10	[CTRL]-[J]	(Line feed)	74	J	138	□	(F10)	202	Ê
11	[CTRL]-[K]		75	K	139	□	(HELP)	203	Ë
12	[CTRL]-[L]	(Löschen)	76	L	140	□		204	Ì
13	[CTRL]-[M]	(RETURN)	77	M	141	□		205	Í
14	[CTRL]-[N]		78	N	142	□		206	Î
15	[CTRL]-[O]		79	O	143	□		207	Ï
16	[CTRL]-[P]		80	P	144	□		208	Ð
17	[CTRL]-[Q]		81	Q	145	□		209	Ñ
18	[CTRL]-[R]		82	R	146	□		210	Ò
19	[CTRL]-[S]		83	S	147	□		211	Ó
20	[CTRL]-[T]		84	T	148	□		212	Ô
21	[CTRL]-[U]		85	U	149	□		213	Õ
22	[CTRL]-[V]		86	V	150	□		214	Ö
23	[CTRL]-[W]		87	W	151	□		215	□
24	[CTRL]-[X]		88	X	152	□		216	Ø
25	[CTRL]-[Y]		89	Y	153	□		217	Ù
26	[CTRL]-[Z]		90	Z	154	□		218	Ú
27	[CTRL]-[[]	(ESC)	91	[	155	□		219	Û
28	[CTRL]-[\]	(hoch)	92	\	156	□		220	Ü
29	[CTRL]-[]]	(runter)	93	]	157	□		221	Ý
30	[CTRL]-[^]	(rechts)	94	^	158	□		222	Þ
31	[CTRL]-[_]	(links)	95	_	159	□		223	ß
32			96	`	160			224	à
33	!		97	a	161	¡		225	á
34	"		98	b	162	¢		226	â
35	#		99	c	163	£		227	ã
36	$		100	d	164	¤		228	ä
37	%		101	e	165	¥		229	å
38	&		102	f	166	¦		230	æ
39	'		103	g	167	§		231	ç
40	(		104	h	168	¨		232	è
41	)		105	i	169	©		233	é
42	*		106	j	170	ª		234	ê
43	+		107	k	171	«		235	ë
44	,		108	l	172	¬		236	ì
45	-		109	m	173	-		237	í
46	.		110	n	174	®		238	î
47	/		111	o	175	¯		239	ï
48	0		112	p	176	°		240	ð
49	1		113	q	177	±		241	ñ
50	2		114	r	178	²		242	ò
51	3		115	s	179	³		243	ó
52	4		116	t	180	´		244	ô
53	5		117	u	181	µ		245	õ
54	6		118	v	182	¶		246	ö
55	7		119	w	183	·		247	□
56	8		120	x	184	¸		248	ø
57	9		121	y	185	¹		249	ù
58	:		122	z	186	º		250	ú
59	;		123	{	187	»		251	û
60	<		124	|	188	¼		252	ü
61	=		125	}	189	½		253	ý
62	>		126	~	190	¾		254	þ
63	?		127	▨	191	¿		255	ÿ

Anhang B: Fehlermeldungen

System-Fehler

-54	Objekt wird schon benutzt | Lock
-53	Objekt existiert schon
-52	Verzeichnis nicht gefunden
-51	Objekt nicht gefunden
-50	Ungültiges Fenster
-49	Unbekannter Fehler 207
-48	Unbekannter Fehler 208
-47	Unbekannter PACKET REQUEST TYPE
-46	Unerlaubter Dateiname
-45	Unbekannter OBJECT LOCK
-44	Objekt-Typ passt nicht
-43	Diskette NOT VALIDATED
-42	Diskette schreibgeschützt
-41	RENAME mit verschiedenen Geräten
-40	Verzeichnis nicht leer
-39	Unbekannter Fehler 217
-38	Gerät nicht bereit
-37	SEEK-Fehler
-36	Datei-Kommentar zu umfangreich
-35	Diskette voll
-34	Datei ist gegen Löschen geschützt
-33	Datei ist schreibgeschützt
-32	Datei ist lesegeschützt
-31	Keine DOS-Diskette
-30	Keine Diskette im Laufwerk
-29	Unbekannter Fehler 227
-28	Unbekannter Fehler 228
-27	Unbekannter Fehler 229
-26	Unbekannter Fehler 230
-25	Unbekannter Fehler 231
-24	Keine weiteren Verzeichnis-Einträge

GFA-BASIC-Fehler

0	Division durch Null
1	Überlauf
2	Zahl nicht Integer | -2147483648 .. 2147483647
3	Zahl nicht Byte | 0 .. 255
4	Zahl nicht Wort | -32768 .. 32767
5	Quadratwurzel nur für | positive Zahlen
6	Logarithmen nur für | Zahlen größer Null
7	Unbekannter Fehler
8	Speicher voll
9	Funktion oder Befehl | noch nicht möglich
10	String zu lang | max. 32767 Zeichen
11	Kein GFA-BASIC-3.03-Programm
12	Programm zu lang | Speicher voll | NEW
13	Kein GFA-BASIC-Programm | EOF - NEW
14	Feld zweimal dimensioniert
15	Feld nicht dimensioniert
16	Feldindex zu groß
17	Dim zu groß
18	Falsche Anzahl Indizes
19	Procedure nicht gefunden
20	Label nicht gefunden
21	Bei Open nur erlaubt: | "I"nput "O"utput "R"andom | "A"ppend "U"pdate
22	File schon geöffnet
23	File # falsch
24	File nicht geöffnet
25	Falsche Eingabe, keine Zahl
26	Fileende erreicht | EOF
27	Zu viele Punkte für | Polyline/Polyfill | maximal 128
28	Feld muß eindimensional sein
29	Anzahl Punkte größer als Feld
30	Merge - Kein ASCII-File
31	Merge - Zeile zu lang - Abbruch
32	==> Syntax nicht korrekt | Programmabbruch
33	Marke nicht definiert
34	Zu wenig Data
35	Data nicht numerisch
36	Unbekannter Fehler 036
37	Diskette voll
38	Befehl im Direktmodus | nicht möglich
39	Programmfehler | Kein Gosub möglich
40	Clear nicht möglich in | For-Next-Schleifen oder | Proceduren
...	

GFA-BASIC-Fehler

	...
41	Cont nicht möglich
42	Zu wenig Parameter
43	Ausdruck zu komplex
44	Funktion nicht definiert
45	Zu viele Parameter
46	Parameter falsch | keine Zahl
47	Parameter falsch | kein String
48	Open "R" - Satzlänge falsch
49	Zu viele "R"-Files (max. 31)
50	Kein "R"-File
51	Unbekannter Fehler 051
52	Fields größer als Satzlänge
53	Unbekannter Fehler 053
54	GET/PUT Field-String | Länge falsch
55	GET/PUT Satznummer falsch
56	Unbekannter Fehler 056
...	
59	Unbekannter Fehler 059
60	Sprite-String-Länge falsch
61	Fehler bei RESERVE
62	Menu falsch
63	Reserve falsch
64	Pointer falsch
65	Feldgröße < 256
66	Kein VAR-Array
67	ASIN/ACOS falsch
68	Falsche VAR-Type
69	ENDFUNC ohne RETURN
70	Unbekannter Fehler 070
71	Index/Parameter zu groß
72	Zu viele Einträge
73	Kann Fenster/Screen nicht öffnen
74	Fenster fehlt
75	Library nicht geöffnet
76	Unbekannter Fehler 076
...	
83	Unbekannter Fehler 083
84	Object. Fehler
85	Drucker-Fehler
86	Unbekannter Fehler 086
87	Unbekannter Fehler 087
	...

GFA-BASIC-Fehler

	...
88	Speicheranforderung fehlgeschlagen
89	Sprite-Fehler
90	Fehler bei Local
91	Fehler bei For
92	Resume (next) nicht möglich |Fatal, For oder Local
93	Stapel-Fehler
94	Unbekannter Fehler 094
...	
99	Unbekannter Fehler 099
100	GFA-BASIC Version 3.03 D|` Copyright 1986-1988|GFA Systemtechnik GmbH
101	Unbekannter Fehler 101
102	Unbekannter Fehler 102
103	Unzureichender Speicher
104	Task-Tabelle voll
105	Unbekannter Fehler 105
...	
119	Unbekannter Fehler 119
120	Argument-Zeile ungültig|oder zu lang
121	Kein ausführbares Programm
122	Ungültige residente Library|beim Laden angesprochen

Neue Fehlermeldungen ab Version 3.5

Anmerkung

Fehlermeldungen, die sich mit der neuen Version nicht geändert haben, wurden in dieser Liste nicht aufgezählt. Bitte sehen Sie dazu in der Standard-Liste der GFA-BASIC-Fehlermeldungen nach, die Sie auch in diesem Buch finden.

Doppelt aufgeführte Meldungen haben sich in der neuesten Version geändert!

11	Kein GFA-BASIC-3.5-Programm
	...
76	Matrizenoperationen nur für ein- oder zweidimensionale Felder
77	Matrizen haben nicht die gleiche Ordnung
78	Vektorprodukt nicht definiert
79	Matrizenprodukt nicht definiert
80	Scalarprodukt nicht definiert
81	Transposition nur für zweidimensionale Matrizen
82	Matrix nicht quadratisch
83	Transposition nicht definiert
	...
86	FACT/COMBIN/PERMUT nicht definiert
	...
100	GFA-BASIC Version 3.5 D|c Copyright 1986-1989|GFA Systemtechnik GmbH
	...
109	Guru Meditation
110	Bus Error
111	Address Error
112	Illegal Instruction
113	Zero Divide
114	CHK Instruction
115	TRAPV Instruction
116	Privilege Violation
117	Trace
118	Line A Emulator
119	Line F Emulator
120	Argument-Zeile ungültig oder zu lang
121	Kein ausführbares Programm
122	Ungültige residente Library beim Laden angesprochen

Anhang C: DOS-Fehlermeldungen

103	**insufficient free store** (Der Speicherplatz reicht nicht aus) Ein Programm kann nicht geladen werden, weil eines oder mehrere Programm-Segmente nicht im Speicher untergebracht werden können.
104	**task table full** (Die Prozeß-Tabelle ist voll) Im CLI können maximal 20 Prozesse vom AmigaDOS verarbeitet werden. Diese Zahl ist erreicht.
120	**argument line invalid or too long** (Die Parameter der Eingabezeile sind ungültig oder zu lang) Das AmigaDOS kann die eingegebenen Parameter nicht auswerten, oder aber es wurden mehr als 255 Zeichen eingegeben.
121	**file is not an object module** (Es handelt sich bei dem File nicht um ein ladbares Programm) Nur Programm-Files können durch einfache Eingabe des Namens gestartet werden. Daten-Files sind nur vom Programm aus ladbar. Dieser Fehler tritt auf, wenn man eine Batch-Datei nur über ihren Namen starten will. Das ist erst ab Version 1.3 über das Flag S möglich.
122	**invalid resident library during load** (Die residente Library ist während des Ladens ungültig) Während des Ladens einer residenten also im Speicher befindlichen Library ist ein Fehler aufgetreten.
202	**object in use** (Das angesprochene File-Objekt wird gerade benutzt) Der Zugriff auf das Directory, File oder Programm wird nicht gestattet, wenn ein anderer Task darauf einen exclusiven Zugriff angemeldet hat. Sehen Sie dazu auch die Open()-Funktion und File-Handles.

203 object already exists

(Das Objekt existiert bereits)

Das Erstellen des Objektes mit dem gewünschten Namen ist nicht möglich, weil darunter schon ein anderes Objekt existiert. Man kann nicht ein Verzeichnis und File mit gleichem Namen innerhalb eines Directories einrichten.

204 directory not found

(Das Verzeichnis existiert nicht)

Das angegebene Verzeichnis existiert nicht unter diesem Pfad.

205 object not found

(Das Objekt konnte nicht gefunden werden)

Der von Ihnen angegebene Name weist auf eine Datei hin, die in diesem Verzeichnis nicht zu finden ist. Entweder stimmt der Name oder der Pfad nicht.

206 invalid windows

(Die Angaben für das Window liegen nicht im erlaubten Bereich)

Beim Öffnen eines neuen Fensters müssen die Koordinaten angegeben werden. Diese dürfen nur in dem über den Screen definierten Bereich liegen. Außerdem muß das Format eingehalten werden. Achten Sie auch auf die Schrägstriche!

209 packet requested type unknown

(Der Typ eines Packets ist dem System nicht bekannt)

Bei der Programmierung einer Nachrichten-Kommunikation ist ein Packet-Typ falsch eingegeben worden. Im System kann dieser Fehler nicht auftreten. Nur wenn Sie eigene Programme schreiben, wird hier wohl der Packet-Typ nicht stimmen oder aber dieses Packet an ein Gerät gesendet, das diese Nachricht nicht kennt.

210 invalid stream component name

(Der Name des Daten-Kanals ist ungültig)

Jeder Daten-Kanal muß einen eindeutigen Namen haben, damit ohne Probleme die Nachricht verschickt werden kann. Dies ist hier nicht der Fall.

211 invalid object lock

(Der Lock zu einem File oder Directory ist fehlerhaft)

Der Lock auf ein File ist nicht mehr gültig. Das kann auftreten, wenn ein Lese-Zugriff von einem Schreib-Zugriff übertrumpft wird und das File nicht mehr in der Art existiert.

212 object not of required type

(Das angesprochene Objekt ist nicht vom geforderten Typ)

Es fand eine Verwechselung zwischen File und Directory statt.

213 disk not validated

(Die Diskette ist nicht validiert)

Der Zugriff auf eine Diskette wird nur gestattet, wenn diese validiert ist. Wurde eine Diskette eingelegt und ist sie nicht validiert, so wird automatisch der Disk-Validator geladen und gestartet. Konnte er nicht geladen werden, erhält man diese Nachricht.

214 disk write-protected

(Die Diskette ist schreibgeschützt)

Ein Schreibzugriff auf diese Diskette ist nicht möglich!

215 rename across device attempted

(Das Umbenennen über Devices hinweg ist nicht gestattet)

Es ist mit Rename durchaus möglich, den gesamten Pfad-Namen zu verändern. Dies beinhaltet aber nicht das Device, was ja einem Kopieren gleichkäme.

216 directory not empty

(Das Verzeichnis ist nicht leer)

Zum Löschen eines Verzeichnisses muß zuerst der Inhalt selbst gelöscht werden, bevor das Verzeichnis vernichtet werden kann.

218 device not mounted

(Das Device ist nicht in das System eingebunden)

Es wird versucht, ein Device anzusprechen, das sich nicht in der System-Liste befindet.

219 seek error

(Die Seek-Funktion kann nicht ordnungsgemäß arbeiten)

Der Seek-Befehl hat falsche Parameter erhalten, so daß er nicht ordnungsgemäß arbeiten kann.

220 comment too big
(Der Kommentar ist zu lang)
Ein Kommentar zu einem File oder Directory darf maximal 80 Zeichen lang sein. Dies wurde hier überschritten.

221 disk full
(Die Diskette ist voll)
Es gibt keinen freien Speicher mehr auf der Diskette, so daß der Schreibvorgang abgebrochen werden mußte.

222 file ist protected from deletion
(Das File ist gegen Löschen geschützt)
Dieses File oder Directory kann nicht gelöscht werden.

223 file ist protected from writing
(Das File ist gegen Schreiben geschützt)
Dieses File kann nicht überschrieben werden.

224 file ist protected from reading
(Das File ist gegen Lesen geschützt)
Dieses File kann nicht gelesen werden.

225 not a DOS disk
(Es ist keine DOS-Diskette)
Die im Laufwerk liegende Diskette hat kein DOS-übliches Format.

226 no disk in drive
(Keine Diskette im Laufwerk)
Es liegt keine Diskette im angegebenen Laufwerk.

232 no more entries in directory
(Es gibt keine weiteren Einträge im Verzeichnis)
Beim Auslesen eines Verzeichnisses mit der Funktion ExNext() tritt diese Meldung nach dem letzten Eintrag auf.

Anhang D: Verzeichnis der GFA-BASIC-Befehle

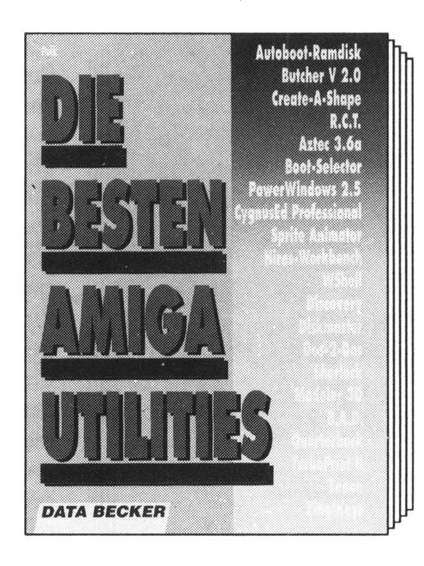
Anhang E: Quellenhinweis

Die Idee des Abbildungsbeispiels auf Seite 470 wurde aus dem Buch "Lineare Algebra mit dem Computer" von E. Lehmann entnommen (ISBN 3-519-02511-6) und anstelle in Pascal in GFA-BASIC programmiert.

Stichwortverzeichnis

Kleine Helfer, die sich sehen lassen können.

Utilities sind immer eine feine Sache – je nach Programm können sie die Arbeit am Rechner erheblich erleichtern, oder auch schon 'mal den einen oder anderen Fehler wieder gutmachen. Einziger Haken: „Dank" der meist unzureichenden Beschreibung ist man oft nicht in der Lage, den gesamten Leistungsumfang des jeweiligen kleinen Helfers zu überblicken. Daher dieses Buch: Die besten Amiga Utilities. Und tatsächlich bietet Ihnen dieser Band eine detaillierte Beschreibung der beliebtesten und stärksten Hilfsprogramme – von der Installation über die Bedienung bis hin zu nützlichen Tips. Hier die Utility-Hitliste: Diskmaster, Butcher V2, Discovery, der Editor CygnusEd Professional, Quarterback, Aztec C-Compiler, Power Windows, Create-A-Shape, Zenon und Zing!Keys. Eben alles, was in der Amiga-Utility-Szene Rang und Namen hat, wird in diesem Band besprochen. Umfassend, detailliert und mit vielen praktischen Anwendungshinweisen. Die besten Amiga-Utilities – das „Handbuch" zu Ihren Hilfsprogrammen.



Polk
Die besten Amiga Utilities
403 Seiten, DM 39,-
ISBN 3-89011-108-4





Das große GFA-BASIC-Buch

Seit einiger Zeit ist GFA-BASIC für den Amiga erhältlich. Endlich ist auch der Compiler da. Zu dem umfangreichen BASIC gehört auch ein umfangreiches Buch, in dem jeder Befehl detailliert behandelt wird. Dieses Buch liefert keine nackte Befehlsübersicht, sondern Informationen, Hinweise und Tips in Hülle und Fülle. Anhand zahlreicher Beispielprogramme lernen Sie das leistungsfähige GFA-BASIC spielend kennen. Um dem Inhalt des Buches abzurunden, ist dem neuen GFA-BASIC-Compiler ein eigenes, ausführliches Kapitel gewidmet.

Aus dem Inhalt:

- Das Editor-Menü
- Variablen- und Speicherorganisation
- Strukturierte Programmierung
- Maus- und Menüabfrage im eigenen Programm
- Intuition-Programmierung
- Abfrage von Ereignissen
- Programmierung von Menüs mit Unterpunkten
- Nutzung der ROM-residenten Libraries
- und vieles mehr ...

Durch viele anschauliche Beispielprogramme lernen Sie schnell den Umgang mit GFA-BASIC und dem neuen GFA-BASIC-Compiler, der Ihre Programme erst richtig schnell macht.

